IT-Security

Usable Privacy with ARP Spoofing

Bachelorarbeit

zur Erlangung des akademischen Grades

Bachelor of Science in Engineering (BSc)

eingereicht von

Tobias Dam

is131003

im Rahmen des

Studienganges IT-Security an der Fachhochschule St. Pölten

Betreuung

Betreuer: Dr. Markus Huber, MSc

St. Pölten, August 12, 2016 _____ _____

(Unterschrift Verfasser/in) (Unterschrift Betreuer/in)

*

Fachhochschule St. Pölten GmbH, Matthias Corvinus-Straße 15, A-3100 St. Pölten,T: +43 (2742) 313 228, F: +43 (2742) 313 228-339, E:office@fhstp.ac.at, I:www.fhstp.ac.at

Ehrenwörtliche Erklärung

Ich versichere, dass

- ich diese Bachelorarbeit selbständig verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und mich sonst keiner unerlaubten Hilfe bedient habe.
- ich dieses Bachelorarbeitsthema bisher weder im Inland noch im Ausland einem Begutachter/einer Begutachterin zur Beurteilung oder in irgendeiner Form als Prüfungsarbeit vorgelegt habe.
- diese Arbeit mit der vom Begutachter/von der Begutachterin beurteilten Arbeit übereinstimmt.

Der Studierende/Absolvent räumt der FH St. Pölten das Recht ein, die Bachelorarbeit für Lehre- und Forschungstätigkeiten zu verwenden und damit zu werben (z.B. bei der Projektevernissage, in Publikationen, auf der Homepage), wobei der Absolvent als Urheber zu nennen ist. Jegliche kommerzielle Verwertung/Nutzung bedarf einer weiteren Vereinbarung zwischen dem Studierenden/Absolventen und der FH St. Pölten.

Ort, Datum *(Unterschrift Autor/Autorin)*

Abstract

The ascending utilisation of third-party frameworks and web content provides developers with a straightforward way to integrate other platforms and advertisements. Nonetheless, the prevalence of third-party content enables organisations to collect an abundance of user data as well as to analyse the users' browsing behaviour and the feeble submission policies of some advertisement companies are a recipe for malware distribution. This bachelor thesis presents a method for using the security attack ARP spoofing to increase users' Internet privacy. The implemented approach routes the traffic of all devices on the local network through the users upribox by crafting ARP packages and therefore impersonating the gateway for those devices. The solution was developed as a module, which installs the Python ARP spoofing Apate and also handles configuration, for the existing project upribox. In conjunction with the newly developed module, the upribox is able to filter web traffic, protect the user from web analytic techniques and block advertisements which may be used to distribute malware, for any device on the network without requiring the reconfiguration of those devices. Furthermore, potential enhancements of the created approach as well as the implemented upribox module are discussed.

Keywords: ARP spoofing, Internet privacy, network security, upribox

Contents

1. Introduction

Topics of this bachelor thesis are the utilisation of ARP spoofing for positive applications as well as the enhancement of users' Internet privacy. The topics were chosen, since the importance of protecting privacy is steadily growing. Due to the rapidly increasing trend to integrate third-party content and advertisements into the own website or application, evermore privacy and IT security threats for example data collection trough web analytics or malvertising arise [1], [2].

1.1. Problem

Since several years, a very large and still rising number of websites and other systems that process web information use frameworks, features and website content of third-party organisations.
This trend in developing websites with the help of other organisations' contributions and integrating other third-party content provides the possibility to enrich user experience and provide new and unprecedented functionality. Some of those functions are the easy implementation of advertising, web analytics and the integration of social networks [1].
Unfortunately, this trend also entails some drawbacks, especially relating to user privacy and security. Due to third-party content being present on many different websites, those are able to retrieve a lot of data like the users Internet browsing behaviour, the users' habits and interests.
Mayer et al. [1] describe the data, organisations may be able to retrieve, as follows:
"Web browsing history is inextricably linked to personal information. The pages a user visits can reveal her location, interests, purchases, employment status, sexual orientation, financial challenges, medical conditions, and more. Examining individual page loads is often adequate to draw many conclusions about a user; analyzing patterns of activity allows yet more inferences." [1]
Another noteworthy threat, which arose from the increased utilization of third-party advertisement frameworks, is malvertising. The Semantec Internet Security Threat Report 2016 mentions that malvertising is on the rise since 2015 and provides attackers with the possibility of infecting user with malicious code through advertisements [2, p. 22]. Many providers of advertisement networks do not apply strict requirements for submitting ads, therefore creating an easy way of distributing malware and infecting

users.

As a result of various affairs relating to security issues that have been announced publicly, an increasing number of people are worried about their privacy:

"The study also found that two thirds (64%) of users are more concerned today about online privacy than they were compared to one year ago." [3]

The percentage stated above shows, that the majority of Internet users is aware of the rising threat to Internet privacy and many of them use traditional IT security solutions to protect themselves.

The most famous of those traditional security solutions is antivirus software. A very common method of operation of Antivirus software is signature-based detection, which searches for specific patterns in programs that identify malicious software.

Another option to protect oneself against malvertising is the utilisation of ad blocking software, which prevents the website from loading advertisements and therefore guards against the infection with malware.

Alongside those conventional measures to increase privacy and IT security in general, there are also rather maverick options, which use techniques well-known relating to security attacks.

In October 2005, Mark Russinovich of Sysinternals discovered a special variant of DRM[1] software, that enforced active protection of music files [4]. Active protection means that a program has to be installed on the users' computer, which afterwards interferes with attempts to copy the music files. The XCP (Extended Copy Protection) software that was installed by CDs shipped by Sony-BMG used a second component, which tried to hide its own and the anti-copy protection software's existence. Therefore Sony-BMG used a so-called rootkit to hide the DRM program and hinder users from illegally copying music files.

Many tools like antivirus and parental control software analyse the Internet traffic of users in order to prevent attacks for instance drive-by downloads, unwanted advertisement or protect children's online activities [5]. A majority of these tools also tries to protect encrypted HTTPS traffic. They intercept the users' traffic by inserting an active man-in-the-middle (MITM) proxy, which separately encrypts traffic from the proxy to the client and from the proxy to the server. By bypassing the original security of the encrypted traffic by using the MITM proxy and afterwards analysing and again encrypting the data, this software is able to add an additional layer of security and control.

Another established security attack with also positive applications is ARP poisoning, which is explained in the chapters 2.1, 2.2 and 2.3. ARP spoofing can be used to intercept traffic and impersonate other computers. Because of this characteristic, ARP spoofing by various software for failover clusters for

[1] Digital Rights Management

instance Heartbeat.

Heartbeat is a software package for High Availability Clusters running Linux, which provides the ability to failover from one computer to another [6, p. 111].

If the primary server of a cluster fails, another server of the cluster has to take over the workload [6, p. 122], [7, p. 30]. Therefore, the new server has to inform the other hosts on the network, that it is now using the shared IP address of the cluster. This announcement is done through gratuitous ARP requests, which are broadcasted to all hosts [6]. These packets contain the shared IP address as sender and target protocol address as well as the MAC address of the new server as protocol sender address. After receiving the request, the host updates its ARP cache with the entry and is now able to communicate with the new server of the cluster.

A second application, which makes use of ARP spoofing to provide failover redundancy, is OpenBSD CARP [8]. The Common Address Redundancy Protocol (CARP) allows a group of hosts to share the same IP address and is usually used by high-availability firewalls. The master host sends regular heartbeat messages to the backup hosts in order to signal a normal state of operation. If the master host fails, one of the backup hosts will take over the master role. This procedure and also the advanced load balancing feature of CARP use crafted ARP packages in order to inform other hosts and clients.

The ascending threats against users Internet privacy through third-party content and advertisements as well as the feasibility of using rather maverick methods in order to establish IT security led to the question whether such methods could be used to increase the users Internet privacy, especially methods that are currently rather uncommon for protecting privacy like ARP spoofing.

1.2. Pivotal Questions and Main Contributions

Due to the problem described in section 1.1 the aim of this bachelor thesis is to answer the following pivotal question:

Can ARP spoofing be a valid mechanism to defend privacy?

In order to answer aforementioned pivotal question as well as to find and implement a proper solution, following sub questions need to be considered.

- **How is it possible to increase Internet privacy with the help of ARP spoofing?**
- **How can a found approach be implemented?**
- **Which possibilities except ARP spoofing could be used for increasing the Internet privacy?**

The main contributions of the bachelor thesis include:

- The development of a concept in order to use ARP spoofing for increasing the users' Internet privacy.
- The practical implementation of the previously developed concept.
- The definition of needed steps in order to further enhance the solution and increase privacy.

1.3. Structure of the Thesis

Section 1 explains the problem, states the pivotal question and main contributions. In section 2 the fundamentals of the Address Resolution Protocol, special use cases, attacks and mitigation techniques are explained. Additionally, other used technologies needed for finding a solution are presented. Section 3 provides information on the design of the solution. The practical implementation of the design and the environment to test the solution are described in section 4. Furthermore, the used research methods are described. Section 5 answers the research questions as well as presents future work and section 6 deals with the conclusion.

2. Theoretical Background & State-of-the-Art

This chapter describes several topics and theories like ARP, ARP spoofing and defence measures, which were needed in order to solve the problem presented at section 1.1.

2.1. ARP

The Address Resolution Protocol (ARP), as specified by Plummer [9], is used to translate addresses of variable length of a specific protocol to 48 bit Ethernet addresses, also called Media Access Control (MAC) addresses.
ARP aims to build a bridge between the logical addresses of different layer 3 protocols and physical addresses which are used on layer 2. Therefore, it provides a simple mapping of <protocol, address> pairs into Ethernet addresses with the help of a translation table, also called ARP cache [9], [10].
The intended workflow of this protocol corresponds to the following example. Assuming that host A wants to send an IP packet to host B and both hosts are on the same network, host A has to translate the IP address of host B into an Ethernet address. If the host does not know the Ethernet address of the other host, it has to broadcast an ARP request packet.
The ARP packet consists of several fields, including:

- the IP address of host A as sender protocol address,
- the MAC address of host A as sender hardware address,
- the IP address of host B as target protocol address,
- the protocol type,
- the opcode, which specifies if the packet is a request or a response.

In an ARP request the target hardware address field is empty, because the sender does not know the hardware address of the target. The host, which receives the ARP request first, checks whether it understands the specified hardware type and protocol type. The second step is to update any existing entry for the

received <protocol, sender protocol address> pair in the translation table. In case there is no entry for this pair in the translation table, a new entry is added. After this action is performed, the receiving host checks the opcode field of the packet and decides if it needs to send an ARP response. An ARP response is created by swapping the sender hardware field and sender protocol field with the according target fields of the request, adding the sender hardware address and changing the opcode to ares_op$reply. The response packet is then sent through an unicast to host A [9].

It should be noted that the protocol specification of the Address Resolution Protocol highlights the two following aspects:

"Notice that the <protocol type, sender protocol address, sender hardware address> triplet is merged into the table before the opcode is looked at." [9]

"Notice also that if an entry already exists for the <protocol type, sender protocol address> pair, then the new hardware address supersedes the old one." [9]

This means, that a host will update an entry or create a new entry in the translation table regardless whether the ARP packet type is a request or a response. "This is on the assumption that [sic] communcation is bidirectional; if A has some reason to talk to B, then B will probably have some reason to talk to A." [9]

A special use case of ARP is the so called gratuitous ARP. This feature consists of an ARP request with the sender protocol address and the target protocol address set to the IP address of the sending host. This can be used to determine whether there is another host with the same IP address configured or to notify other hosts that the Ethernet address of the host has been changed, which seldom occurs [10].

2.2. ARP Spoofing

Due to the fact of ARP being a stateless protocol and therefore processing unrequested ARP replies, as mentioned in section 2.1, it is an easy task for an attacker to poison the ARP cache of other devices.

It is possible to get a host to add a fake entry to the ARP cache by sending a crafted ARP request or reply packet with the desired source hardware address to the victim. The victim will update its translation table or add a new entry, since the opcode of the ARP packet is checked after the packet has been processed [9], [11, pp. 10].

"Hence an attacker only needs to send a spoofed ARP request (inherently broadcasted) to poison the cache of all the hosts in a LAN." [11, p. 11]

ARP replies are distinguished between "normal" ARP replies and unsolicited responses [11, p. 10], which are not anticipated by a host.

ARP spoofing is often used by attackers to perform man-in-the-middle (MITM) attacks. A traditional

MITM attack using ARP spoofing works by corrupting the ARP cache of two victims, with the result that host A communicates with the attacker if it wants to send information to host B and vice versa. Additionally, the attacker forwards the received packets to the original recipient, in order to tunnel the data between the victims and to be able to read and manipulate the data [12].
As mentioned by Samineni, Barbhuiya & Nandi [12] this method "[...] comes with an inherent drawback that the attacker has to leave his own (though temporary/crafted) MAC address on the target nodes." [12]
Samineni et al. [12] also specify two variations of the traditional MITM attack using ARP spoofing:

- the "Stealth man-in-the-middle (SMITM) attack"
- the "Semi-stealth man-in-the-middle (SSMITM) attack".

The SMITM attack works similar to the common MITM attack, except that the attacker states the broadcast MAC address FF:FF:FF:FF:FF:FF as the sender hardware address instead of his own MAC address, which enables the attacker to stay unrecognised [12]. As a result the two victims are sending packets via broadcast each time they want to communicate. Due to the information being broadcasted and the attacker only receiving a copy of the sent information, it is not possible to manipulate the packets on-the-fly.
In the case of one host being secured and filtering ARP packets with the broadcast MAC address set as sender hardware address, it is possible to use the SSMITM attack [12]. During this attack only the unsecured host receives the crafted ARP packet with the broadcast address. The secured host gets a "traditionally" forged ARP reply, which contains the hardware address of the attacker instead of the unsecured hosts address. If the unsecured host A communicates with the secured host B, the broadcasted packets of host A are discarded by host B. The attacker redirects the information from host A, which was send via broadcast, to host B using his MAC address. Due to the attacker also redirecting received packets from host B to host A, he is able to tunnel the communication and manipulate the forwarded data. The advantage of this method is the fact, that the physical address of the attacker is only known to one victim and therefore appears ordinary.
The disadvantage of the two methods mentioned by Samineni et al. [12] is the simple detectability by checking whether an IP address is translated to the broadcast address.

2.3. Defence Measures Against ARP Spoofing

In order to protect systems from MITM attacks using ARP spoofing various mitigation and prevention approaches have been developed.

The simplest method to prevent entries of the ARP cache being changed by malicious ARP packets is to configure static ARP entries [13, pp. 186-188]. In spite of the fact that these static entries are manually set along with the desired hardware and MAC addresses, this method is not applicable for dynamic networks[1] and large networks, because of the high maintenance effort needed to keep the ARP tables updated.

2.3.1. S-ARP

The "Secure Address Resolution Protocol" (S-ARP) is a more sophisticated solution, which provides an message authentication for ARP reply messages using public key cryptography [14]. This is achieved by appending an additional header field containing authentication information and the digital signature of the S-ARP reply as payload to the original ARP packet, which means that S-ARP is backward compatible to ARP. A "Authoritative Key Distributor" (AKD) has to be set up in the network, which contains a certificate from each host with the according public key and IP address. When a host wants to verify a received ARP reply and is not already in possession of the public key of the sender, the host is able to receive the needed certificate from the AKD.
The disadvantage of this method was mentioned by Bruschi et al. [14] in section "7. Conclusions and Future Work": "Another issue concerns the elimination of the single point of failure represented by the AKD." [14]

2.3.2. TARP

Another approach, which is based on cryptographic authentication is the "Ticket-based Address Resolution Protocol" (TARP) [15]. TARP performs message authentication by appending an attestation, a so called ticket, to each ARP reply packet as additional payload. These tickets contain several fields including the MAC address and IP address of a host, the expiration time of the ticket and the signature of the ticket. TARP requires the setup of a "Local Ticket Agent" (LTA), which issues and signs tickets using its private key for each host. A host that receives a TARP reply is able to verify the signature of the ticket with the LTAs public key and check if the received <IP address, MAC address> pair is valid. The drawback of TARP is the vulnerability to active host impersonations[2] and DoS attacks through ticket flooding.

[1] A dynamic network is a network where hosts are often added and removed.

[2] Active host impersonation in this case means to impersonate a victim by spoofing the MAC address and using a previously captured ticket. This is possible until the ticket expires.

2.3.3. MR-ARP

The “MITM-Resistant Address Resolution Protocol” (MR-ARP) consists of a long-term <IP address, MAC address> mapping table in addition to the ARP cache and voting-based conflict resolution [16]. The long-term table is used to save accepted <IP address, MAC address> mappings for a certain amount of time[3].

When a host receives an ARP request with an <IP address, MAC address> pair, that is not registered in the ARP cache, the long-term table is checked. If it already contains a matching entry, the lifetime of this entry is extended. When the long-term table has a conflicting entry, the host sends 50 ARP request packets to the stored MAC address. The old entry is preserved after receiving an ARP reply, otherwise the new mapping is added to the long-term table. If the received <IP address, MAC address> pair is registered in neither the ARP cache nor the long-term table, the voting-based conflict resolution process is started. This process consists basically of asking other hosts for their entries to the received <IP address, MAC address> pair. When the host does not receive any replies, it registered the received mapping in the ARP cache and in the long-term table. Otherwise the MAC address which obtained over 50 per cent of the vote replies is registered.

2.3.4. KARP

A recent attempt to solve the ARP spoofing problem is the “Kerberos Secured Address Resolution Protocol” (KARP) which integrates the Kerberos protocol[4] into ARP [18]. KARP consists of multiple KARP clients and one KARP server. Each client stores its IP address as an ID as well as two shared secret keys with the server.

The server has two main parts, which handle the authentication:

- the “Authentication Service” (AS)
- the “Ticket Granting Service” (TGS).

The AS provides “Single Sign On” (SSO) through authenticating the clients and so called “Ticket Granting Tickets” (TGT). The TGS issues “Destination Tickets” that authorize a client to send a KARP request to another client.

The procedure of KARP is described in following example:

Client A is currently not logged in and wants to send a KARP request to client B. Initially, client A has

[3] The default value for the timer is 60 minutes.

[4] “Kerberos provides a means of verifying the identities of principals, (e.g., a workstation user or a network server) on an open (unprotected) network." [17]

to send a "Ticket Granting Ticket Request" containing his ID to the AS in order to log in. If client A is authorized, the AS responds with a TGT. In the next step, client A sends a "Destination Ticket Request" to the TGS, which consists of the destination ID (of client B), the TGT and an authenticator (IP and MAC address of client A). The TGS answers to valid requests with an encrypted "Destination Ticket", which proves the identity of client A through its IP and MAC addresses. Now client A is able to broadcast a new KARP request with the received "Destination Ticket" to all other clients. Only client B is able to decrypt the ticket, because the ticket was requested for client B as destination. Client B compares the specified IP and MAC addresses in the KARP header with those from the ticket and sends a KARP reply if the request was valid. The reply message is encrypted with a newly generated ticket for client A, so that client A is also able to verify the KARP message.
Bakhache et al. [18] mention that in course of their performance test KARP performed the address resolution considerably faster than the other solutions S-ARP [14] and TARP [15].

2.4. Usable Privacy Box (upribox)

The Usable Privacy Box is a project of the Institute of IT Security Research [19] at St. Pölten University of Applied Sciences, which was conducted by Dr. Markus Huber, MSc. The project was funded by "netidee 2014 der Internet Foundation Austria (IPA)" and aims to be a tool for increasing the privacy of Internet users while focussing on intuitive usability [20].
The primary hardware component of the upribox is a Raspberry Pi, an affordable, credit card-sized computer which is plugged into the users' modem. Additionally needed hardware is listed on the official GitHub repository [21].
After the successful setup of the upribox, the user can connect their devices to the automatically created wireless network. The upribox automatically filters unencrypted HTTP[5] traffic and protects users from miscellaneous web analytic techniques, web beacons and similar threats to Internet privacy [20].
Additional services are the provision of an OpenVPN[6] server, which allows to connect to the upribox from virtually anywhere and the possibility to route the whole traffic through Tor[7].
The upribox software mainly consists of two parts:

- the Django web interface

[5] "The Hypertext Transfer Protocol (HTTP) is an application-level protocol for distributed, collaborative, hypermedia information systems." [22]
[6] An open-source application framework which implements virtual private networks to create secure connections.
[7] Tor "[...] is a distributed overlay network designed to anonymize TCP-based applications like web browsing, secure shell, and instant messaging." [23, p. 1]

- and multiple Ansible roles.

The web interface of the upribox uses the Django web framework, as mentioned in section 2.6 and is written in Python. It allows the user to intuitively make several configuration changes like modifying the SSID and the password of the created wireless networks.
The setup of the upribox including the installation of the web interface and several configuration changes are done using the Ansible Automation Framework, which is described in section 2.5. This part of the software is structured in so-called roles which provide flexibility, easy separation of production and development code and simple extensibility.
The actual filtering of the web traffic is done via utilising Privoxy[8] and Dnsmasq[9]. Several files containing blocking rules are used to indicate which DNS requests or web contents should be blocked.

2.5. Ansible

Ansible is an open-source IT automation framework for configuring and orchestrating computers and is developed by Red Hat Inc [24]. The Ansible framework is based on Python and uses YAML[10] for the description of various tasks.
Ansible uses playbooks to define the configuration and the tasks, which it should carry out. Those can be written in one single file or be split up in multiple folders and files, so-called roles. A role can perform several tasks, for instance installing updates, configuring services or copying files, can define handlers which are called as soon as a specific condition is met and can also define numerous variables for adapting the configuration for specific systems or environments.
Because of security considerations, Ansible uses OpenSSH, one of the most peer-reviewed open-source components, for the data transport. The system is also decentralized as it only needs existing credentials to access remote machines and is also able to use various authentication systems like Kerberos or LDAP [24].

2.6. Django

Django is an open-source web development framework developed by the Django Software Foundation [25]. The framework is written in Python and follows the well-established Model-View-Controller software pattern. Django's design is highly modular as the developer is able to structure the project into sev-

[8] Privoxy is a web proxy with advanced filtering functions.
[9] Dnsmasq is a simple DNS and DHCP server.
[10] YAML Ain't Markup Language (YAML) is a human-readable data exchange format.

eral application, which are performing different tasks. Another advantage of Django is the circumstance, that it focusses on security and it is developed to be scalable and therefore uses available resources. A “web page" provided by Django consists of a model, a view and a template part. The model is used to manage data of the web application, whereas the view provides the logic for processing HTTP requests and returning a response. The template defines the information which should be presented to the user.

2.7. Scapy

Scapy is a multipurpose packet manipulation tool developed by Philippe Biondi [26], [27]. The program is written in Python, internally uses libpcap[11] or WinPCap on Windows and is able to perform various tasks regarding network traffic manipulation. Those tasks include forging and decoding packets of various network protocols as well as sniffing for specific packets with user defined reactions. Scapy enables the user to define each layer of a packet, stack those layers upon another and set values of protocol fields as desired. It does not restrict the user by enforcing values, that are valid per protocol specification and also allows to forge a packet combining protocols that usually are not used together or in a different order.

2.8. Redis

Redis is an open-source, in-memory key value store written in the C programming language and is developed by Salvatore Sanfilippo and Redis Labs [28]. It functions as a NoSQL database[12] and is commonly utilised as a cache or message broker. Redis supports various data structures, for instance strings, hashes as well as sets and despite of working with in-memory datasets, this data is stored persistently on the disk by default. Another function provided by Redis is to subscribe or publish to a channel and therefore to react to certain events. Such an event is the “expired event”, which is sent after a key reaches the specified time to live (TTL) [29].

2.9. Neighbor Discovery Protocol

The Neighbor Discovery (ND) for IP version 6 (IPv6), as specified by Narten et. al. [30], is used to perform various tasks for IPv6, which were previously performed by the Address Resolution Protocol (ARP), as described in section 2.1, the ICMP Router Discovery (RDISC) and ICMP Redirect (ICMPv4). Those tasks include:

[11] Implementation of the pcap api for capturing network traffic for Unix-like systems.

[12] NoSQL databases use different techniques to store data than for example the tables of relational databases.

- the Router and prefix discovery,
- the Address Resolution,
- the Neighbor Unreachability Detection,
- the Redirect Function.

The router and prefix discovery function of ND enables the location of neighouring IPv6 routers and the distribution if the address prefixes and configuration parameters used for the stateless address auto-configuration [30, p. 38]. Hosts of the network can send router solicitation messages to receive router advertisements, which are used for the address configuration of the host and the default router selection [30, pp. 53-58].
Another part of the Neighbor Discovery Protocol is the address resolution and neighbor unreachability detection function, which is composed of neighbor solicitation and neighbor advertisement messages [30, pp. 22-25]. The way of functioning of this part of ND is very similar to the Address Resolution Protocol used by IPv4. Therefore the main aim of the address resolution is to provide a mapping between the IPv6 address and the link-layer address. The neighbor solicitation and neighbor advertisement messages consist of several fields, including:

- the IP source address,
- the IP destination address,
- the ICMP type,
- the ICMP code,
- the ICMP checksum,
- the ICMP target address,
- the optional ICMP source/target link-layer address.

In case host A needs to send data to another host B on the same network and does not know the link-local address of host B, it sends a neighbor solicitation messages to the other host as well as creates a neighbor cache entry in the INCOMPLETE state [30, pp. 22-25], [30, pp. 61-67]. The IP source address field of the message contains the IPv6 address of host A and the IP destination address field is set to the IPv6 address of host B or a solicited-node multicast address[13] corresponding to the IPv6 address of host B.

[13] IPv6 multicast address computed of a node's unicast or anycast address [31, p. 16].

The ICMP type of the neighbor solicitation message is 135 and the ICMP code is 0. Host A needs to set the ICMP target address to the IPv6 unicast address of host B and specify its own link-layer address in the ICMP source link-layer address field. The specification of the own link-layer address is only mandatory, when the IP destination address field contains a multicast address.
After host B receives the neighbor solicitation message from host A, it answers by sending a neighbor advertisement message with its own IPv6 address as IP source address and the IPv6 address of host A as IP destination address [30, pp. 22-25]. The ICMP type of this message is 136 and the ICMP code is 0. The ICMP target address is set to the same value of the corresponding field of the neighbor solicitation message and the target link-layer address is filled with the link-layer address of host B.
Additionally, there are three ICMP flags for the neighbor advertisement message:

- the router flag (R), which indicates if the sender is a router,
- the solicited flag (S), which indicates if this message is a response to a neighbor solicitation message,
- the override flag (O), which indicates whether the advertisement should override existing neighbor cache entries.

Host B also updates an existing entry in its neighbor cache entry or creates a new entry in state STALE [30, pp. 61-67]. Once host A receives the valid advertisement it updates its neighbor cache entry and sets the state to REACHABLE.
The redirect function consists of redirect messages, which are sent by routers to inform hosts about a better first-hop router or that a destination is a neighbor of the host [30, p. 73]. A host that receives such a redirect messages modifies its destination cache, in order to send subsequent traffic to the destination specified in the redirect message [30, p. 76].

2.10. Internet Group Management Protocol

The Internet Group Management Protocol version 3 (IGMPv3), as specified by Cain et. al.[32], enables systems to report their multicast group memberships to neighboring multicast routers. IGMP version 3 distinguishes between two types of IGMP version 3 messages:

- Membership queries with the type number 0x11,
- Membership report with the type number 0x22.

IGMPv3 membership queries are sent by multicast routers to learn the multicast reception state of hosts on the network [32, pp. 8-11]. The most important fields of the membership query packets are the type field, which contains the value 0x11, the group address, the number of sources and the source address fields. The group address is set to the IP multicast address of the multicast group, which should be queried. In case the enquirer is only interested in the memberships of specific hosts, those hosts are listed in the source address fields and the number of sources is set accordingly. There are also additional fields, including the checksum, the suppress router-side processing (S) flag, the querier's robustness variable (QRV) and the querier's query interval code (QQIC), which are used to manage the transmission and processing behaviour of messages.
Depending of the values of specific fields, there are three types of queries [32, pp. 12]:

- The "General Query", which is sent to learn the multicast group memberships of all hosts on the network. The group address field and the number of sources are set to zero, while an IP destination of 224.0.0.1 is used.
- The "Group-Specific Query", that is used to query the members of a single multicast group. The group address field and the IP destination contain the IP address of the multicast group of interest. Additionally, the number of source is set to zero.
- The "Group-and-Source-Specific Query" is similar to the "Group-Specific Query", but is also limited to the hosts of interest, which are specified through the source address fields. The number of source is set accordingly to the number of source address fields.

IGMP membership queries are answered by sending membership reports, which contain a type of 0x22, a checksum, the number of group records and the according group records [32, pp. 12-17]. The most important information of a group record is the record type and the multicast address the record applies to. The record type indicates different states, for instance that the host is listening to or filtering a specific multicast group or that it wants to start or stop listening to a specific multicast group.

3. Design

This chapter presents the theoretical design of the solution, created to solve the pivotal question and the sub questions mentioned in section 1.2.

3.1. General

Due to the functioning principle of ARP spoofing, spoofing individual clients on the network by transmitting forged ARP packets and subsequently redirect traffic generated by those clients, this technique alone is not sufficient for defending user's Internet privacy directly. Rather than merely redirecting the data, which is received and transmitted by the user's devices, the information also needs to be refined or filtered in order to prevent threats against privacy. Therefore, a solution utilising ARP spoofing to protect Internet privacy furthermore requires a functionality to process the Internet traffic transmitted by other devices.

A constantly enhanced open-source project which provides aforementioned necessary feature is the upribox, as described in section 2.4. The upribox aims to be an easy to use, zero configuration device, which helps users to protect their Internet privacy by filtering the Internet traffic of their own devices. Filtering the data is accomplished by processing the data with Privoxy and custom filter rules. In order to use those features, users have to connect their devices to the wireless network created by the upribox. This essential user interaction somewhat contradicts with the zero configuration ideology of the project, though it could be prevented by automatically redirecting the entire network traffic through the upribox, thus enabling automated filtering of every device.

For the purpose of answering the research question, mentioned in section 1.2 and to resolve the usability issue of the upribox, the approach was realised as an extension for the upribox.

The deployment of the upribox is performed with the help of Ansible, specified in section 2.5, and organising the different functions of the project is done via Ansible roles. The developed approach makes use of the existing upribox source code and is integrated into the project as a new Ansible role.

This Ansible role named "arp" performs two major tasks:

- making preparations in order to install the ARP spoofing daemon,

- and the installation of the Apate daemon itself.

The needed preparations include creating necessary directories, copying startup scripts for the daemon, installing necessary additional software and adding or modifying configuration files. The redirection of the network traffic is done by the Apate ARP spoofing daemon, whose structure is explained in the section 3.2.

3.2. Apate

The daemon, which handles the spoofing of the clients on the network, is implemented as a well-behaved Python Unix daemon, called Apate[1]. This program knows the functions *start*, *stop* and *restart* and is automatically started and stopped with the help of startup scripts.
The configuration file of Apate is located at the path `/etc/apate/config.json` and stores the configuration options in the JSON format, because of its easy readability for humans and to be consistent with Ansible's local facts, which are also stored using the JSON format.
The configuration file determines the values of the following configuration options:

- `pidfile`, the path of the pidfile of the daemon,
- `logfile`, the path of the logfile of the daemon,
- `interface`, the network interface that should be used to perform ARP spoofing,
- `stdout`, the path of the file which contains the content of standard output,
- `stderr`, the path of the file which contains the content of standard error,
- `mode`, the definition of the mode of operation that Apate should use.

After the daemon is started, it begins to check the validity of the configuration file and initialises general properties, needed in order to spoof the clients of the network. Those initialisation activities include gathering information about the gateway and the network configuration of the spoofing device. The next step after the daemon finished the preparations, is the spoofing of the network devices itself. Apate knows two different modes of operation for this task, which are explained in section 3.3. In case the daemon is stopped, the original ARP cache entries are restored in order to allow the clients on the network to communicate normally.

[1] Apate is the Greek daemon of deceit [33, pp. 2670].

3.3. Modes of Operation

The Apate ARP spoofing daemon knows two different ways of functioning, which can be configured to be one of the following values via the configuration option `mode`:

- `holistic`, which enables the holistic spoofing mode as described in section 3.3.1.
- `selective`, which enables the selective spoofing mode as described in section 3.3.2.

Each mode consists of a regular reoccurring part, that spoofs the ARP cache entries of devices on the network with forged ARP packets. For the purpose of additionally handling devices, that are brought online while the first part is not active or clients, which ask for the gateway, the second part acts as listener for ARP packets. The third part restores the "normal" network operation of all hosts on the network and the gateway.

3.3.1. Holistic Spoofing

The holistic spoofing mode of Apate makes use of broadcast gratuitous ARP request packets for manipulating the ARP table entries of the user's devices. In the course of this the Ethernet destination is set to the broadcast address `ff:ff:ff:ff:ff:ff` as well as the ARP hardware destination field. Both, the Ethernet source field and the ARP hardware source field, are filled with the hardware address of the network interface specified by the `interface` configuration option. In order to create a valid GARP request packet, the protocol source and the protocol destination are set to the IP address of the gateway and the opcode is set to 1, being a request. Each host on the network receives this broadcasted packet and updates its existing ARP table entries.
In this mode the daemon generates gratuitous ARP request packets for every possible client on the network and sends them to the gateway. The packets are unicast requests with the Ethernet source set to the hardware address of the gateway and the ARP protocol source and protocol destination being the IP address of the possible host. As mentioned in section 2.1, if a client receives an ARP request it only updates existing entries and does not create new ones dissimilar to the actions after receiving ARP replies. Therefore, the gateway's ARP tables is not flooded by creating unnecessary entries.
Due to the generation of packets for every possible host and the spoofing of clients by broadcast GARP requests, the need for a host discovery part and the possibility to store the information of discovered hosts disappears. This implies lower resource requirements, than the selective spoofing mode of Apate, which performs such tasks.
A drawback of this mode of operation is the huge number of packets, which have to be created to spoof the gateway, if the upribox is used in a big network, for example a network with a subnetmask in CIDR-

Notation of `/8`. The generation and also the transmission of the vast amount of packets takes a rather long time, which causes a delay of spoofing of the devices. Therefore, it is likely that the network devices will not be spoofed properly. As a consequence of this behaviour, this mode is more suitable for rather small networks (`/24`).

Another important part of Apate's holistic spoofing mode is the listener, that is used to react to certain incoming packets. This component listens for incoming ARP requests and distinguishes between unicast requests, which are intended for the upribox itself and broadcast ARP requests, which are transmitted to or received from the gateway. After processing a gratuitous ARP request, which changes an existing ARP cache entry, some systems ask the "new" device via unicast ARP requests, if it really has the stated IP address. If those systems do not receive an answer in a certain amount of time, they ask for the device with the stated IP address by sending a broadcast ARP request. This would cause the real gateway to send an answer to the spoofed device, which could result in "unspoofing" it. Therefore, the listener handles incoming ARP requests with an Ethernet destination set to the hardware address of the upribox, so as to answer the requests before any broadcast packets are sent.

The other type of processed packets are broadcast ARP requests with a protocol source or destination being the IP address of the gateway. The fields of those packets are used to spoof both the transmitter and the receiver of the request or to be more accurate, the gateway and the involved host. The ARP reply intended for manipulating the transmitter is mostly created by swapping source and destination fields of the sniffed packet. The Ethernet destination is set to the former Ethernet source, furthermore the protocol destination and the hardware destination are filled with the according original source values. In order to impersonate the initial target, the IP address of the target is used as protocol source and the hardware address of the network interface of the upribox is the hardware source.

The former receiver is spoofed by generating an unicast ARP reply. The Ethernet destination and the hardware destination contain the hardware address of the original target, which has to be learned by sending a proper broadcast ARP request. The hardware address of the upribox is used as hardware source, whereas the protocol source and protocol destination use the former values.

Due to some systems not accepting ARP replies immediately after receiving the first ARP reply of the gateway, for instance after the configuration of the IP address via DHCP, the transmission of the packets, generated in response to incoming broadcast requests, is delayed for a specific amount of time.

The third part of the holistic spoofing mode preserves the "normal" network operation by restoring the original ARP table entries. A GARP broadcast request is generated to tell network hosts the address of the real gateway. The Ethernet destination as well as the hardware destination contain the broadcast address, whereas the protocol source and protocol destination use the IP address of the gateway. The

hardware source matches the hardware address of the gateway. In order to correct the modified entries of the gateway, ARP request packets are generated for every possible client on the network. This results in the existing clients sending ARP replies to the gateway, exposing their real addresses.

3.3.2. Selective Spoofing

A disadvantage of the holistic spoofing mode is the vast amount of packets, which are necessary to spoof the gateway and impersonate every possible client on the network. Generating and sending these packets takes plenty of time, which might hamper the successful redirection of the network traffic. The selective spoofing mode of Apate addresses this problem by only generating packets for existing hosts. This is enabled through the utilisation of a database to save entries for existing devices persistently. Each entry stores information about the network address according to the IP address of the device, the IP address itself, the host's hardware address and if spoofing should be enabled. The selective spoofing mode requires one or more host discovery methods for detecting available devices in contrast to the holistic spoofing mode. Similar to the other mode of operation, this mode also consists of a periodic part, a listener and another part, which handles the return to the "normal" network operation.

The main task of the reoccurring part is the creation of two unicast ARP replies for each device entry in the database. The first packet spoofs the client, by using the stored hardware address of the device as value of the Ethernet destination and the hardware destination, whereas both source fields are filled with the upribox's hardware address. The protocol destination contains the stored IP address of the device and the protocol source is set to the gateway's IP address.

The second packet manipulates the arp entry of the gateway by impersonating the device. Both hardware source fields contain the hardware address of the upribox and both hardware destination fields are set to the hardware address of the gateway. This time, the protocol source contains the stored IP address of the device and the protocol destination is set to the gateway's IP address.

The functionality of the listener corresponds to the respective part of the holistic spoofing mode and adds the creation of device entries. In case the listener receives a unicast ARP request, which is intended for the upribox, it stores a device entry for the transmitting device. An entry is created for the transmitting device along with an entry for the receiving device, if a broadcast ARP request destined to or received from the gateway is processed. Despite of the holistic mode only sniffing for incoming ARP requests, this part also saves device entry for the sender after ARP replies are obtained. The spoofing of the newly added network devices is enabled per default.

As opposed to the second part of Apate's holistic mode, the selective mode is not limited to incoming ARP requests and also processed incoming IGMP report messages. As further explained in section 3.4,

IGMP general query messages are used as an additional host discovery method. Clients on the network answering the IGMP general query answer by sending an IGMP report message stating their multicast group membership. This answer packet contains the needed information to generate a device entry for the discovered host. Furthermore, the newly added device and the gateway are spoofed using layer two and three data of the received packet.
The first task of the third part, the restoration of the original network behaviour, is performed similarly to the holistic spoofing mode. The ARP table entries of the clients are corrected via broadcast GARP requests. Due to the utilisation of the database, the daemon only has to create packets for existing clients during the second task. Unlike the holistic spoofing mode, the gateway's entries are modified directly by forging ARP reply packets for known hosts. The Ethernet destination and hardware destination are set to the hardware address of the gateway, while the hardware source contains the stored hardware address of the device. Additionally, the protocol source is filled with the device's IP address and the protocol destination is the IP address of the gateway.

3.4. Host Discovery Methods

As mentioned in section 3.3.2, the selective spoofing mode makes use of a database in order to store entries for existing network devices. Those entries are created via the listener by sniffed incoming packets. For this reason host discovery methods are utilised, so that hosts on the network transmit these processible packets.
Alongside the traditional host discovery method for local networks, there are also other rather maverick techniques.

3.4.1. Traditional approach

A well-known and established host discovery approach used on local networks is to perform an ARP scan. This method is executed by generating ARP broadcast requests for every possible network host. Existing clients are answering with ARP replies in response to those packets, which are afterwards processed by the listener. In case of the daemon being used in a rather big network (`/8`), the generation of a vast number of packets takes a long time, as already explained in section 3.3.1. This problem might cause a delay during the host discovery, but does not interfere with the spoofing of already known devices.

3.4.2. IGMP general query requests

In order to avoid the aforementioned problem, IGMP general queries can be used to discover network hosts. Therefore, IGMP general queries with a type value of `0x11` and the group address field set to `224.0.0.1`, which is the multicast address including all hosts on the local network, are sent. This causes network devices, which are subscribed to at least one multicast group, to answer with IGMP report messages. Those report message packets contain the information needed to create device entries inside their layer two and layer three fields.

A drawback of this host discovery method is the rather high unreliability. Hosts which are not subscribed to any multicast groups, are not answering the general query requests and hence can not be discovered using this method. Also some devices might not answer general queries and only respond to "Group-Specific Queries". In order to mitigate this undesirable behaviour, the host discovery method could ask for memberships of often used multicast addresses. One possibly suitable multicast address is `224.0.0.252` used by the Link-local Multicast Name Resolution (LLMNR), which is used to provide name resolution on the same local link [34, pp. 1-4]. Another potentially useful multicast address is `224.0.0.251` of the Multicast Domain Name Server protocol, also providing name resolution on the local link [35, pp. 1-6].

4. Evaluation and Results

This section describes the research methodology and presents the specific implementation of the solution of the research question. In the following section, the structure as well as source code of the upribox extension are explained.

4.1. Methodology

The approach was implemented as an Ansible role and as an extension for the existing upribox source code. Several changes of the existing upribox source code were needed to allow the filtering of the spoofed and redirected network traffic. The chosen programming language for developing the Apate ARP spoofing daemon is Python, because most of the original source code is also written in this language. Furthermore, Scapy, which is described in section 2.7, is used for forging and transmitting packets, since it is a well-established project relating to those tasks. The developed approach was constantly tested during development, inside a dedicated network with access to the Internet.

4.2. Changes of Existing upribox Ansible Roles

As mentioned in section 2.4, the filtering of the web traffic on the upribox are handled by Privoxy and Dnsmasq. In order to provide the full privacy enhancing functionality not only to hosts connected to the wireless networks, but also to hosts on the local wired network, some configuration options needed to be changed.

The original configuration of Dnsmasq does not enable listening for DNS requests on the network interface “eth0”. The following line in the configuration template `roles/dns/templates/dnsmasq.j2` has to be replaced, to also provide DNS services for spoofed clients:

```
1 except-interface=eth0
```

Simply uncommenting the line is sufficient for fulfilling this purpose:

```
1 #except-interface=eth0
```

Because DHCP services must not be enabled for the network interface “eth0”, this feature is disabled by adding following line to `roles/dns/templates/dnsmasq.j2`:

```
1 no-dhcp-interface=eth0
```

Another additional line for the same configuration template is needed, because the change, that enables DNS services on the interface “eth0”, might create a vulnerability to DNS amplification attacks[1]. If the interface “eth0” is publicly exposed to the Internet, it may be used as an open DNS forwarder and therefore for an DNS amplification attack. This issue is prohibited by adding `local-service` to the file `roles/dns/templates/dnsmasq.j2`. Due to this configuration option, Dnsmasq will only accept DNS queries from local subnets [37].

Due to the configuration option `listen-address 0.0.0.0:8118`, Privoxy on the upribox listens on every network interface on port `8118` for requests. The following lines of the Privoxy configuration file `roles/privoxy/templates/config` restrict the access to both upribox wireless networks and the OpenVPN network:

```
1 permit-access  192.168.55.0/24
2 permit-access  192.168.56.0/24
3 permit-access  192.168.155.0/24
```

The access for those three networks needs to be preserved, but also the network, which interface “eth0” belongs to, has to be included. Because the IP configuration of this interface is dynamically retrieved through DHCP and so as not to provide services to the public Internet, the private network address spaces, as specified by Rekhter et ak. [38], are granted access rights to access Privoxy.

```
1 permit-access  10.0.0.0/8
2 permit-access  172.16.0.0/12
3 permit-access  192.168.0.0/16
```

DNS requests of spoofed network clients pass the upribox, but are not processed by the Dnsmasq service of the upribox. They are just forwarded to the configured DNS server of the clients and therefore those clients do not benefit from the privacy enhancing DNS blacklist feature and are unable to access the upribox web interface. Thus, following iptables rule was appended to the “nat” table’s “PREROUTING” chain inside the `roles/iptables/templates/iptables.upribox.ipv4`, which rewrites passing by DNS requests, so that they are processed by the upribox.

```
1 -A PREROUTING -i eth0 -p udp -m udp --dport 53 -j REDIRECT --to-ports 53
```

[1] DNS amplification attacks use recursive DNS servers to direct DNS traffic to a specific target and use the fact, that small requests can trigger large responses [36, p. 3].

4.3. New Ansible Role

The newly created Ansible Role “arp” performs several tasks, including the installation of the Apate ARP spoofing daemon, making necessary preparations and changing other configurations.

The deployment of the new role via Ansible is enabled by adding the following line to the roles section of `upriboxes.yml` and also `local.yml`:

```
1   - { role: arp, tags: ["arp"] }
```

This change enables the manual deployment of the role as well as the automated deployment during the update process and associates the tag “arp” with the eponymous role.

The new Ansible role consists of a directory “arp” inside the folder `roles` and has the following structure:

```
1   +---arp
2   |   +---files
3   |   |   \---apate
4   |   |       |   apate.py
5   |   |       |   requirements.txt
6   |   |       |
7   |   |       \---lib
8   |   |               apate_redis.py
9   |   |               daemon_app.py
10  |   |               extended_runner.py
11  |   |               misc_thread.py
12  |   |               sniff_thread.py
13  |   |               util.py
14  |   |               __init__.py
15  |   |
16  |   +---handlers
17  |   |       main.yml
18  |   |
19  |   +---tasks
20  |   |       apate_state.yml
21  |   |       main.yml
22  |   |
23  |   \---templates
24  |       |   config.json
25  |       |   logrotate.j2
26  |       |
27  |       \---init
```

```
28 |               apate
29 |               apate.service
```

The directory `files` contains the source code files of the Apate ARP spoofing daemon, which are copied onto the upribox by the "arp" role. These files are further explained in section 4.4.
The directory `tasks` includes the Ansible playbook as `main.yml` and the file `apate_state.yml`, which is used to check whether local facts are available and apply those if that is the case. The provided variable `apate_enabled` indicates if Apate should be enabled at startup. Thereby the content of the file `apate_state.yml` is as follows:

```
1 ---
2 - set_fact:
3     apate_enabled: "{{ default_settings.apate.general.enabled if not (ansible_local
          is defined and ansible_local.apate is defined and ansible_local.apate.
          general is defined) else ansible_local.apate.general.enabled | default(
          default_settings.apate.general.enabled) }}"
```

The `main.yml` file contains several important tasks, which are performed by this Ansible role. The following first part of the file includes the aforementioned variable `apate_enabled` and `other_env`, which is used to remove logfiles of the other environment. Additionally, the Apate daemon is copied to `/opt/apate`.

```
1  ---
2  - include: ../../common/tasks/other_env.yml
3  - include: apate_state.yml tags=toggle_apate
4
5  - name: create working directory for apate daemon
6    file: path=/opt/apate state=directory recurse=yes mode=0771 owner=root group=root
7
8  - name: copy the apate files
9    copy: src=apate/ dest=/opt/apate owner=root group=root mode=0774
10   notify: restart apate
```

The next step is to make the Apate daemon start after boot, if the variable `apate_enabled` is yes.

```
1 - name: copy apate init script
2   template: src=init/apate dest=/etc/init.d/apate owner=root group=root mode=0755
3   notify: restart apate
4
5 - name: copy apate service file
6   template: src=init/apate.service dest=/etc/systemd/system/apate.service owner=root
        group=root mode=0755
```

```
7   notify: restart apate
8   register: service_file
9
10 - name: systemctl daemon-reload
11   shell: /bin/systemctl daemon-reload
12   when: service_file.changed
13
14 - name: configure apate service
15   service: name=apate state='{{ "started" if apate_enabled|bool else "stopped" }}'
         enabled='{{ apate_enabled|bool }}'
16   tags:
17     - toggle_apate
```

The following tasks copy the configuration file of Apate and remove the logfiles of the other environment, for example the development environment if production is active, as well as provide a logrotate configuration file. Furthermore, additional software, as listed below, is installed:

- `python-virtualenv`, which is used to install the dependencies of Apate,
- `tcpdump`, which provides functionality used by Scapy,
- `redis-server`, which is used as key-value database by Apate's selective spoofing mode.

```
1  - name: create apate config dir
2    file: path=/etc/apate state=directory recurse=yes mode=0771 owner=root group=root
3
4  - name: copy apate config file
5    template: src=config.json dest=/etc/apate/config.json owner=root group=root mode
         =0755
6    notify: restart apate
7
8  - name: install virtualenv, tcpdump
9    apt: name={{ item }} state="{{ apt_target_state }}" force=yes update_cache=yes
         cache_valid_time="{{ apt_cache_time }}"
10   with_items:
11     - python-virtualenv
12     - tcpdump
13     - redis-server
14
15 - name: install requirements to virtualenv
16   pip: requirements=/opt/apate/requirements.txt virtualenv=/opt/apate/venv
17   notify: restart apate
```

```
18
19 - name: remove log files from other environment
20   file: path={{other_env.default_settings.log.general.path}}/{{other_env.
         default_settings.log.apate.subdir}} state=absent
21
22 - name: modify logrotate.d entry
23   template: src=logrotate.j2 dest=/etc/logrotate.d/apate mode=0644
```

The last part of the file `main.yml` enables the Redis server at startup and modifies the configuration file, so that keyspace event messages for the expiration of keys are emitted.

```
1 - name: change keyspace event notification of redis-sever
2   lineinfile:
3     dest: /etc/redis/redis.conf
4     regexp: '^notify-keyspace-events'
5     line: 'notify-keyspace-events "Ex"'
6   notify: restart redis
7
8 - name: enable redis server
9   service: name=redis-server enabled=yes
```

The file `main.yml` inside the directory `handlers` provides tasks, which can be triggered by the playbook, if certain tasks performed changes to the target. The following tasks restart either the Apate daemon or the Redis server, if according files have changed.

```
1 ---
2 - include: ../tasks/apate_state.yml tags=toggle_apate
3
4 - name: restart apate
5   service: name=apate state={{"restarted" if apate_enabled|bool else "stopped"}}
6
7 - name: restart redis
8   service: name=redis-server state=restarted
```

The files inside the folder `templates/init` handle the startup of the Apate daemon. The file `apate` is a init script, which creates necessary directories containing the pidfile and the logfiles as well as provides the options `start`, `stop` and `restart`. The script makes use of the Linux Standard Base (LSB) comment conventions for init scripts[2] and hereby determines services, which are required to start before Apate, like networking services. The content of the file is available in the appendix A.4.

[2]The comment conventions are used ensure that init scripts are started and stopped correctly corresponding to their requirements [39, pp. 394-396].

In order to provide the same functionality for systems using `systemd` to manage services, the service file `apate.service`, in appendix A.5, also specifies the aforementioned options and defines dependencies of the service. Additionally, the option `Restart=on-failure` causes `systemd` to restart the daemon in case of a crash.

The file `templates/logrotate.j2`, which is copied to `etc/logrotate.d/apate`, defines the actions logrotate should take for Apate's logfiles. The files are rotated weekly or after the reach a size of 10 MB. The file is available in appendix A.6.

The file `templates/config.json` serves as the configuration file of the Apate ARP spoofing daemon and contains the options already mentioned in section 3.2. The file contains the available options in the JSON format as follows:

```
1 {
2     "pidfile": "{{ default_settings.apate.pid.dir }}/{{ default_settings.apate.pid.
          file }}",
3     "logfile": "{{ default_settings.log.general.path }}/{{ default_settings.log.
          apate.subdir }}/{{ default_settings.log.apate.logfiles.logname }}",
4     "interface": "eth0",
5     "stdout": "{{ default_settings.log.general.path }}/{{ default_settings.log.apate
          .subdir }}/{{ default_settings.log.apate.logfiles.stdout }}",
6     "stderr": "{{ default_settings.log.general.path }}/{{ default_settings.log.apate
          .subdir }}/{{ default_settings.log.apate.logfiles.stderr }}",
7     "mode": "{{ default_settings.apate.mode }}"
8 }
```

This file does not already contain the actual values of the options, because these values can differ depending on the active environment. These values are defined inside the `group_vars/all.yml` file of the according environment, either `environments/production` or `environments/development`, and are automatically inserted during the deployment via Ansible, when it is copied to `/etc/apate/config.json`. These variables are defined inside the “apate" sections of aforementioned files as well as in appendix A.8 and A.9.

4.4. Apate

Following the design of the Apate ARP spoofing daemon, presented in section 3.2, the directory `files/apate` contains all necessary source code files for implementing the well-behaved Python Unix daemon. All source code files are provided under appendix A.10.

4.4.1. Requirements

Several third-party software and python libraries are needed in order to provide the functionality of the Apate daemon. As specified in section 4.3, the playbook `main.yml` installs the additional software packages `python-virtualenv`, `tcpdump` and `redis-server`.

The package `python-virtualenv` provides the opportunity to separately install Python packages in specific versions, while not interfering with different instances needed by other applications. Therefore `python-virtualenv` was chosen for the separation of the Python dependencies of Apate.

Installing `tcpdump` enables Scapy to compile BPF[3] filters, which are used inside Scapy's `sniff` functionality [41].

Apate's selective sniffing mode requires a Redis server as key value datastore for storing several device entries. Redis was chosen, because of its support for various data structures as well as the possibility to subscribe to messages, as mentioned in section 2.8. Furthermore, the information, that needs to be stored, does not require relationships or a tabular structure, which was also a reason for relying on this NoSQL database.

In addition to the third-party software, various Python libraries are needed by the Apate daemon:

- `redis`,
- `hiredis`,
- `scapy`,
- `python-daemon`,
- `netaddr`,
- `netifaces`.

The `redis` Python library provides an interface for the Redis key value store by supporting most of the native Redis commands. It also implements functionality for subscribing to channels and listening for specific messages. This library uses the `hiredis` package for parsing the values returned by the Redis server.

Scapy, as described in section 2.7, is used for sniffing purposes as well as forging and the dissection of packets by both the holistic spoofing mode and the selective spoofing mode of Apate. This project was chosen due to its diverse capabilities as well as the non-restrictive approach.

[3] The Berkeley Packet Filter (BPF) enables user-level processes to capture packets and uses a register-based filter machine for filtering out packets [40, p. 259].

Apate's daemon behaviour is implemented via the `python-daemon` package, which is conform to the Python Enhancement Proposal (PEP) 3134. This package provides everything necessary to create a well-behaved Unix daemon written in Python. Since this package is well-established and has an alignment with the PEP 3134 guideline, it was selected.

The libraries `netaddr` and `netifaces` provide several networking functionalities, whereas `netifaces` retrieves the hardware and IP addresses of network interfaces and `netaddr` offers functions for processing IP addresses. Those packages are essential for the operation of the Apate ARP spoofing daemon.

All Python requirements are installed during the execution of the playbook `main.yml` to the virtual environment `/opt/apate/venv` using the file `requirements.txt`. The exact versions of necessary libraries and additional dependencies of those are listed inside `files/apate/requirements.txt` in appendix A.10.8.

4.4.2. General Information

As almost every program, the Apate daemon possesses a main part, which is run by the init script or manually, if preferred by the user. This part is written inside the `files/apate/apate.py` file. The full content can be reviewed in appendix A.10.1.

This script specifies the location of the configuration file, as detailed in section 4.3 and 3.2, as well as mandatory configuration options inside the following two constants:

```
1 CONFIG_FILE = "/etc/apate/config.json"
2 """Path of the config file for the Apate ARP spoofing daemon."""
3 CONFIG_OPTIONS = ('logfile', 'pidfile', 'interface', 'stderr', 'stdout', 'mode')
4 """Options that need to be present in the config file."""
```

Inside the `main()` function, a check whether the script runs with root privileges is performed. Apate uses several functionalities, for instance networking services through Scapy, which requires root privileges. Afterwards, the configuration file is loaded via the Python `json` package and the mandatory options are checked. In case that an error occurs during processing the configuration file, the daemon exists with an appropriate error code. The next step is to initialise the logging facility and to determine if Apate should use the holistic spoofing mode or the selective spoofing mode. The last remaining action prepares the `DaemonApp` and finally starts the daemon.

Both modes of operation of the Apate daemon are defined by inheriting from the class `_DaemonApp` located in the file `files/apate/lib/daemon_app.py`. These subclasses define the actual behaviour of the daemon and need to perform several actions, which are used during both modes and are therefore implemented inside the superclass.

The superclass implements following methods:

- `__init__(self, logger, interface, pidfile, stdout, stderr)`,
- `__return_to_normal(self)`,
- `exit(self, signal_number, stack_frame)`,
- `run(self)`.

All methods are described inside the function by using Python documentation strings, including the method parameters and their expected types. The following code shows the documentation of the parameters of the `__init__` method:

```
1 Args:
2     logger (logging.Logger): Used for logging messages.
3     interface (str): The network interface which should be used. (e.g. eth0)
4     pidfile (str): Path of the pidfile, used by the daemon.
5     stdout (str): Path of stdout, used by the daemon.
6     stderr (str): Path of stderr, used by the daemon.
```

The `__init__` method uses aforementioned parameters to initialise the daemon, create logfiles, retrieve information about the IP and hardware addresses of the used network interface as well as the gateway. Additionally, a list, including every possible IP address on the specific network, except addresses like the network address or the broadcast address, is created for further use inside the modes. In case the daemon fails during the initialisation phase, it raises a `DaemonError` to indicate this.
The methods `run` and `__return_to_normal` are just method stubs, which need to be overridden by the subclasses. The first mentioned method determines the main part of the mode of operation, including the way the spoofing of network clients is done. The second method is used to return the "normal" network state by "unspoofing" the clients.
The `exit` method is used by the `python-daemon` library and terminates the daemon after executing `__return_to_normal`.

4.4.3. Holistic Spoofing

The design of Apate's holistic spoofing mode, as specified in section 3.3.1, is implemented with following classes:

- the `HolisticDaemonApp`,
- the `HolisticSniffThread`.

The `HolisticDaemonApp` class is located inside the file `files/apate/lib/daemon_app.py`, which is available in appendix A.10.3, and inherits from the abstract `_DaemonApp.py` class. Expanding the `__init__` method of the superclass, it also initialises an instance of the `HolisticSniffThread`, which is used to sniff incoming ARP packets. This thread is defined to be a daemon thread, meaning it terminates as well if the main thread exits.

Following code defines the way the holistic mode spoofs clients periodically:

```
1  def run(self):
2          # start sniffing thread
3          self.sniffthread.start()
4
5          # generates a packet for each possible client of the network
6          # these packets update existing entries in the arp table of the gateway
7          packets = [Ether(dst=self.gate_mac) / ARP(op=1, psrc=str(x), pdst=str(x))
               for x in self.ip_range]
8          # gratuitous arp to clients
9          # updates the gateway entry of the clients arp table
10         packets.append(Ether(dst=ETHER_BROADCAST) / ARP(op=1, psrc=self.gateway,
               pdst=self.gateway, hwdst=ETHER_BROADCAST))
11         while True:
12             sendp(packets)
13             time.sleep(self.__SLEEP)
```

At first the sniffing thread, which is described below, is started. Afterwards, a list `packets`, containing gratuitous ARP request packets for spoofing the gateway, is created using the list of possible IP addresses created inside the `__init__` method. These packets are used to impersonate network clients and update existing ARP cache entries of the gateway, while not adding new entries for non-existent ones. A single broadcast GARP request packet is appended to `packets`, for spoofing all clients on the network. Every host receives this broadcast and updates its ARP table entry for the gateway with the upribox's hardware address. As the content of these packets does not change, they are only created once and sent every `__SLEEP` seconds.

The constant `__SLEEP` is implemented as a class-level variable and defaults to 20 seconds:

```
1  __SLEEP = 20
2  """int: Defines the time to sleep after packets are sent before they are sent anew.
       """
```

The ARP requests inside `packets` are sent using Scapy's `sendp` method, because the Ethernet layer of the packet is specified manually. This method inserts "logical" values for non-specified field, for instance the Ethernet source field. In this case the hardware address of the upribox is filled in as Ethernet source

address automatically. Scapy's `sendp` method performs best, when multiple packets are sent at once by providing them inside a list. Therefore all ARP requests are stored inside the list `packets` and sent via a single method call.

At termination of the daemon the method `exit` is called, which in turn executes `_return_to_normal`:

```
1 def _return_to_normal(self):
2     # clients gratutious arp
3     sendp(
4         Ether(dst=ETHER_BROADCAST) / ARP(op=1, psrc=self.gateway, pdst=self.gateway,
              hwdst=ETHER_BROADCAST,
5                                         hwsrc=self.gate_mac))
6     # to clients so that they send and arp reply to the gateway
7     sendp(Ether(dst=ETHER_BROADCAST) / ARP(op=1, psrc=self.gateway, pdst=str(self.
          network), hwsrc=self.gate_mac))
```

This method follows the same principle as the `run` method. The network clients are spoofed per broadcast gratuitous ARP request with the IP address and hardware address of the original gateway. The entries of the gateway's ARP cache are corrected by requesting the hosts on the network to send ARP replies to the gateway. This is also done by generating broadcast ARP requests for every possible device.

Sniffing of incoming ARP requests is performed by the `HolisticSniffThread`, which inherits the abstract class `_SniffThread`, a subclass of `threading.Thread`.

The abstract class defines following methods:

- `__init__(self, interface, gateway, mac, gate_mac)`,
- `run(self)`,
- `_packet_handler(self, pkt)`,
- `stop()`.

The `__init__` method initialises several variables storing information about the network interface and the gateway as well as calls the constructor of `threading.Thread`.

The `run` method uses Scapy's `sniff` functionality to start sniffing for specific packets and calling `_packet_handler` for every received packet.

Following class-level variables are used as parameters for the `sniff` method:

```
1 _DELAY = 7.0
2 """float: Delay after which packets are sent."""
3 _SNIFF_FILTER = "arp and inbound"
4 """str: tcpdump filter used for scapy's sniff function."""
```

```
5 _LFILTER = staticmethod(lambda x: x.haslayer(ARP))
6 """function: lambda filter used for scapy's sniff function."""
```

The aforementioned method is called as below:

```
1 sniff(prn=self._packet_handler, filter=self._SNIFF_FILTER, lfilter=self._LFILTER,
      store=0, iface=self.interface)
```

The parameter `prn` defines the method, that should be called after a packet of the specified type is received. The packet is provided as a parameter for further processing. The second argument `filter` specifies a string containing a BPF filter, which is used to filter packets before they are handed to Scapy and provides performance advantages. The value of `_SNIFF_FILTER` is a filter, which only passes incoming ARP packets through. Due to a race condition problem, that may result in receiving packets that should be filtered by the BPF filter, the remaining packets are additionally filtered by a Python method provided via `lfilter`. The constant `_LFILTER` provides a lambda function checking if the packet has an ARP layer. `store=0` indicates, that received packets should be passed to the function in `prn` and not be stored. The interface of the upribox is specified via `iface`.

The method `_packet_handler` is a method stub, which should be overridden by subclasses. In case the thread is not started as a daemon thread, it can be canceled by calling the static `stop` method.

The `HolisticSniffThread` class uses all provided methods of the superclass and only overrides the `_packet_handler` method.

```
1  def _packet_handler(self, pkt):
2          """This method is called for each packet received through scapy's sniff
               function.
3          Incoming ARP requests are used to spoof involved devices.
4
5          Args:
6              pkt (str): Received packet via scapy's sniff (through socket.recv).
7          """
8          # when ARP request
9          if pkt[ARP].op == 1:
10
11             # packets intended for this machine (upribox)
12             if pkt[Ether].dst == self.mac:
13                 # this answers packets asking if we are the gateway (directly not
                       via broadcast)
14                 # Windows does this 3 times before sending a broadcast request
15                 sendp(Ether(dst=pkt[Ether].src) / ARP(op=2, psrc=pkt[ARP].pdst, pdst
                       =pkt[ARP].psrc, hwdst=pkt[ARP].hwsrc, hwsrc=self.mac))
```

```
16
17              # broadcast request to or from gateway
18              elif pkt[Ether].dst.lower() == util.hex2str_mac(ETHER_BROADCAST) and (
                    pkt[ARP].psrc == self.gateway or pkt[ARP].pdst == self.gateway):
19                  # spoof transmitter
20                  packets = [Ether(dst=pkt[Ether].src) / ARP(op=2, psrc=pkt[ARP].pdst,
                         pdst=pkt[ARP].psrc, hwsrc=self.mac, hwdst=pkt[ARP].hwsrc)]
21
22                  # get mac address of original target
23                  dest = self.gate_mac
24                  if pkt[ARP].pdst != self.gateway:
25                      # send arp request if destination was not the gateway
26                      dest = util.get_mac(pkt[ARP].pdst, self.interface)
27
28                  if dest:
29                      # spoof receiver
30                      packets.append(Ether(dst=dest) / ARP(op=2, psrc=pkt[ARP].psrc,
                            hwsrc=self.mac, pdst=pkt[ARP].pdst, hwdst=dest))
31
32                  # some os did not accept an answer immediately (after sending the
                        first ARP request after boot)
33                  # so, send packets after some delay
34                  threading.Timer(self._DELAY, sendp, [packets]).start()
```

The if statement inside line 9 checks if the received packet is a ARP request, because only requests are further processed. The next step is to examine, whether the packet is intended for the upribox. As already explained in section 3.3.1, the listener answers ARP requests asking if the upribox is the gateway via unicast request. In case the packet is a broadcast ARP request and received from or transmitted to the gateway, as inspected in line 18, both the transmitter and the receiver of the packet are spoofed. First, the packet to spoof the transmitter is created using the values of the original packet. By using the hardware address of the upribox as hardware source in the variable `hwsrc`, the daemon impersonates the gateway. In order to spoof the original target, the hardware address has to be resolved by the daemon itself. The method `get_mac` is used to perform this task, as described in section 4.6. After the daemon learned the hardware address, it generates the second packet and sends both after a delay specified in `_DELAY`. Some systems do not accept incoming GARP requests immediately after receiving the first ARP reply of the gateway, which the reason for this delay. This difficulty is described in section 3.3.1.

4.4.4. Selective Spoofing

The implementation of Apate's selective spoofing mode following the design, as presented in section 3.3.2, consists of several classes:

- the `SelectiveDaemonApp`,
- the `SelectiveSniffThread`,
- the `PubSubThread`,
- the `IGMPDiscoveryThread`,
- the `ARPDiscoveryThread`.

The selective spoofing mode implies storing information about existing clients on the network. The classes `IGMPDiscoveryThread` and `ARPDiscoveryThread` retrieve this information. Those classes are described in section 4.4.5.
Similar to the `HolisticDaemonApp` the `SelectiveDaemonApp` inherits the `_DaemonApp` superclass and overrides the inherited methods. The `__init__` method is used to additionally initialise an `ApateRedis` instance, specified in section 4.5 and the four different threads mentioned above. All threads are determined to be daemon threads.
The class also defines a class-level variable, used as delay inside the periodic part:

```
1 __SLEEP = 5
2 """int: Defines the time to sleep after packets are sent before they are sent anew.
      """
```

Essentially, the `run` method of the selective mode attains the same result as the according method of the `HolisticDaemonApp`, albeit the internal actions are different.

```
1  def run(self):
2          self.sniffthread.start()
3          self.arpthread.start()
4          self.psthread.start()
5          self.igmpthread.start()
6
7          # lamda expression to generate arp replies to spoof the clients
8          exp1 = lambda dev: Ether(dst=dev[1]) / ARP(op=2, psrc=self.gateway, pdst=dev
               [0], hwdst=dev[1])
9
10         # lamda expression to generate arp replies to spoof the gateway
```

```
11          exp2 = lambda dev: Ether(dst=self.gate_mac) / ARP(op=2, psrc=dev[0], pdst=
                self.gateway, hwdst=self.gate_mac)
12
13          while True:
14              # generates packets for existing clients
15              # due to the labda expressions p1 and p2 this list comprehension, each
                    iteration generates 2 packets
16              # one to spoof the client and one to spoof the gateway
17              packets = [p(dev) for dev in self.redis.get_devices_values(filter_values
                    =True) for p in (exp1, exp2)]
18
19              sendp(packets)
20              time.sleep(self.__SLEEP)
```

At first all threads are started in order to begin sniffing packets and discovering hosts. The next step is to define two different lamda expressions. The first expression `exp1` defines the creation of an ARP reply packet used to spoof a specific network client. The second expression `exp2` specifies an ARP reply packet to spoof the gateway by impersonating one particular device. Both variables are functions used during the list comprehension inside the loop in line 17. This list comprehension uses the expressions to generate two packets for each device belonging to the current network stored inside the Redis database. The devices are retrieved via the `get_devices_values` method of the `ApateRedis` instance. In contrast to the according method of the `HolisticDaemonApp` class, these packets need to be created anew every time the packets are sent, because devices may have been added or removed.

The method `exit` is called upon termination of the daemon, which in turn executes `_return_to_normal`:

```
1   def _return_to_normal(self):
2           # spoof clients with GARP boradcast request
3           sendp(
4               Ether(dst=ETHER_BROADCAST) / ARP(op=1, psrc=self.gateway, pdst=self.
                    gateway, hwdst=ETHER_BROADCAST,
5                                                hwsrc=self.gate_mac))
6
7           # generate ARP reply packet for every existing client and spoof the gateway
8           packets = [Ether(dst=self.gate_mac) / ARP(op=2, psrc=dev[0], pdst=self.
                gateway, hwsrc=dev[1]) for dev in self.redis.get_devices_values(
                filter_values=True)]
9           sendp(packets)
```

The first part of this method restores the ARP table entries of the network clients the same way as the

HolisticDaemonApp. The second part creates ARP replies containing the correct hardware address of the host as hardware source for every existing device and transmits the packets to the gateway.

As mentioned in section 3.3.2, the selective spoofing mode does not suffer from the spoofing delay originating from the generation of packets for every possible host on the network. Inside the _return_to_normal method as well as in run, existing devices are retrieved from the database and used for packet creation, which results in a more performant procedure.

The second part of the selective spoofing mode is the listener, which is implemented by the SelectiveSniffThread class. Similar to the HolisticSniffThread, it inherits the abstract class _SniffThread. The selective mode uses the host discovery methods explained in section 3.4, therefore the class-level variables _SNIFF_FILTER and _LFILTER are overridden in order to be able to process incoming ARP and IGMP packets.

```
1 _SNIFF_FILTER = "(arp or igmp) and inbound"
2 """str: tcpdump filter used for scapy's sniff function."""
3 _LFILTER = staticmethod(lambda x: any([x.haslayer(layer) for layer in (ARP, IGMP)]))
4 """function: lambda filter used for scapy's sniff function."""
```

Addtionally to the parameters of the __init__ method of the superclass, the constructor also takes the argument redis, which is an instance of ApateRedis. The overridden method _packet_handler examines whether the packet is an ARP packet or an IGMP packet and either calls _arp_handler or _igmp_handler.

Most of the _arp_handler method works similar to the _packet_handler method of the HolisticSniffThread class. In case the received packet is an ARP request and it is intended for the upribox, the transmitter is spoofed after the according packet was created. Additionally, a device entry inside the Redis database is stored for the sending device, using following code:

```
1 self.redis.add_device(pkt[ARP].psrc, pkt[ARP].hwsrc)
```

Thereby, the ApateRedis instance self.redis is used to create a new device entry via the method add_device(ip, mac). The used methods to interact with the Redis database are mentioned in section 4.5. The same method is also called in case the received packet is a broadcast ARP request and transmitted from or destined to the gateway in order to create an entry for the transmitter and another entry for the receiver. This handler also processes incoming ARP replies by adding a device entry for the sender.

Incoming IGMP report messages are processed by _igmp_handler, which generates a device entry and spoofs the transmitter of the IGMP report as well as the gateway.

```
1 def _igmp_handler(self, pkt):
```

```
2         self.redis.add_device(pkt[IP].src, pkt[Ether].src)
3         sendp([Ether(dst=pkt[Ether].src) / ARP(op=2, psrc=self.gateway, pdst=pkt[IP
              ].src, hwdst=pkt[Ether].src),
4                 Ether(dst=self.gate_mac) / ARP(op=2, psrc=pkt[IP].src, pdst=self.
                      gateway, hwdst=self.gate_mac)])
```

4.4.5. Host Discovery Methods

As mentioned in section 4.4.4, the selective spoofing mode utilises the classes `IGMPDiscoveryThread` and `ARPDiscoveryThread` for host discovery purposes. Both classes implement a `__init__` as well as a `run` method and inherit the superclass `threading.Thread`.

```
1  class ARPDiscoveryThread(threading.Thread):
2      """This thread is used to discover clients on the network by sending ARP
           requests."""
3
4      def __init__(self, gateway, network):
5          threading.Thread.__init__(self)
6          self.gateway = gateway
7          self.network = network
8
9      def run(self):
10         sendp(Ether(dst=ETHER_BROADCAST) / ARP(op=1, psrc=self.gateway, pdst=self.
               network))
```

The `__init__` method of the `ARPDiscoveryThread` has the parameters `gateway` and `network` being the IP address of the gateway and the network's IP address. After the initialisation is finished, the thread can be started. The `run` method is executed and sends ARP request packets for every possible host. Corresponding answers are process via the `SelectiveSniffThread`.

The second host discovery thread is the `IGMPDiscoveryThread`, which possesses following class-level variables:

```
1  _IGMP_MULTICAST = "224.0.0.1"
2  """str: Multicast address used to send IGMP general queries."""
3  _SLEEP = 60
4  """int: Time to wait before sending packets anew."""
5  _IGMP_GENERAL_QUERY = 0x11
6  """int: Value of type Field for IGMP general queries."""
7  _TTL = 1
8  """int: Value for TTL for IP packet."""
```

The value of _TTL is used during the creation of the IP packet and _IGMP_GENERAL_QUERY specifies the value of the type field. This thread uses IGMP general queries for discovering hosts and therefore uses a type value of 0x11 and the multicast address 224.0.0.1 as defined in _IGMP_MULTICAST. _SLEEP defines the delay between two executions.

The __init__ method of the class works similar to the same-named method of ARPDiscoveryThread , but takes the IP address and the hardware address of the upribox as additional parameters.

The following source code of the run method, creates an Ethernet layer, an IP layer and the IGMP layer separately and afterwards calls the method igmpize to adept some headers of the Ethernet and IP layer. After this packet has been generated once, it is sent every _SLEEP seconds.

```
1   def run(self):
2       # create IGMP general query packet
3       ether_part = Ether(src=self.mac)
4       ip_part = IP(ttl=self._TTL, src=self.ip, dst=self._IGMP_MULTICAST)
5       igmp_part = IGMP(type=self._IGMP_GENERAL_QUERY)
6
7       # Called to explicitely fixup associated IP and Ethernet headers
8       igmp_part.igmpize(ether=ether_part, ip=ip_part)
9
10      while True:
11          sendp(ether_part / ip_part / igmp_part)
12          time.sleep(self._SLEEP)
```

Incoming answers are again processed by the SelectiveSniffThread.

4.5. Apate Redis

The class ApateRedis located inside files/apate/lib/apate_redis.py manages the Redis database and device entries stored inside the database.

Several constants are used to define the device entries, which database should be used and the expiration time of entries:

```
1  __PREFIX = "apate"
2  """str: Prefix which is used for every key in the redis db."""
3  __DELIMITER = ":"
4  """str: Delimiter used for separating parts of keys in the redis db."""
5  __IP = "ip"
6  """str: Indicator for the IP part of the redis key."""
7  __NETWORK = "net"
8  """str: Indicator for the network address part of the redis key."""
```

```
9  __DB = 5
10 """int: Redis db which should be used."""
11 __TTL = 259200
12 """int: Time after which device entries in the redis db expire. (default=3 days)"""
```

Per default Apate uses database 5 of the Redis server as defined by `__DB`, so as to not interfere with other applications using the default database. In order to avoid entries remaining forever in the database, entries are added with an Time to Life (TTL) specified in `__TTL`. Therefore, entries for devices, that have been removed from the network, are automatically deleted by the Redis server after the TTL expires.
The `ApateRedis` class knows two different types of entries:

- the network entry,
- the device entry.

The network entry uses a Redis set as datastructure, that stores the IP addresses of clients for a specific network. The key of a network entry consists of the Apate prefix `__PREFIX`, the network part indicator `__NETWORK` and the network IP address. Each part is delimited by the value of `__DELIMITER`. For example, the key of the network entry for the network `192.168.20.0/24` would be `apate:net:192.168.0.0`. The datatype set was chosen, because it does not allow repeated members, which is advantageous as at most one entry per IP address should exist.
The device entry's value consists of a simple string, containing the hardware address of the device. The key has the same parts as the network entry as well as the IP part indicator `__IP`, the IP address of the device and the enabled state of the device. These parts are also delimited by the value of `__DELIMITER`. In case a device should be enabled for spoofing, the enabled state is set to 1, otherwise it is set to 0. The key of the device entry for a host enabled for spoofing with the network configuration `192.168.0.5/24` would be `apate:net:192.168.0.0:ip:192.168.0.5:1`.
Because of this specific key format it is easily possible to retrieve all devices of a specific network with an enabled spoofing state. The IP addresses stored inside the network entry just have to be prepended with the key of the network entry and the ip indicator. The resulting key is appended with the enabled state 1. The Redis database returns the hardware address if the device entry exists, meaning that the device is enabled. Otherwise the string `"None"` is returned by the Redis server, which can easily be filtered.
The method `__init__` initialises several variables, when a new `ApateRedis` instance is created.

```
1 def __init__(self, network, logger):
2     self.redis = redis.StrictRedis(host="localhost", port=6379, db=self.__DB)
3     self.network = network
4     self.logger = logger
```

This method requires the network address and the instance of the logging facility as parameters and creates a new `StrictRedis` object, which opens a connection to the local Redis server using the default port and the database specified in `__DB`. This object is used to communicate with the Redis server, for instance adding and removing keys.

The class `ApateRedis` has following public methods to manage and retrieve entries:

```
1  def add_device(self, ip, mac, network=None, enabled=True, force=False):
2      if not self._check_device_disabled(ip, network or self.network) or force:
3          self._add_device_to_network(ip, network or self.network)
4          return self._add_entry(self._get_device_name(ip, network or self.network,
               enabled=enabled), mac)
5
6  def remove_device(self, ip, network=None, enabled=True):
7      self._del_device_from_network(ip, network or self.network)
8      return self._del_device(self._get_device_name(ip, network or self.network,
           enabled=enabled))
9
10 def get_device_mac(self, ip, network=None, enabled=True):
11     return self.redis.get(self._get_device_name(ip, network or self.network, enabled
           =enabled))
12
13 def get_devices(self, network=None):
14     return self.redis.smembers(self._get_network_name(network or self.network))
15
16 def get_devices_values(self, filter_values=False, network=None, enabled=True):
17     # list may contain null values
18     devs = self.get_devices(network=network or self.network)
19     if not devs:
20         return []
21     else:
22         vals = self.redis.mget([self._get_device_name(dev, network or self.network,
               enabled=enabled) for dev in devs])
23         res = zip(devs, vals)
24         # filter "None" entries if filter_values is True
25         return res if not filter_values else [x for x in res if x[1] and x[1] != str
               (None)]  # filter(None, res)
26
27 def get_pubsub(self, ignore_subscribe_messages=True):
28     return self.redis.pubsub(ignore_subscribe_messages=ignore_subscribe_messages)
29
30 def disable_device(self, ip, network=None):
```

```
31      self._toggle_device(ip, network or self.network, enabled=False)
32
33  def enable_device(self, ip, network=None):
34      self._toggle_device(ip, network or self.network, enabled=True)
```

Most of the methods shown above require one or more of the following parameters:

- `ip`, the IP address of the device, that should be managed,
- `mac`, the hardware address of the device, that should be managed,
- `network`, the network address that should be used. If this parameter is not used, the value of `self.network` is used instead.
- `enabled`, determines if the entry should be enabled.

The method `add_device` can be used to create a new device entry. The IP address of the device is automatically added to the corresponding network entry. An enabled entry is only created per default, if there is no disabled entry for the same device inside the database. This behaviour can be disabled by passing `force=True` as parameter.
Should a specific device be removed, the method `remove_device` can be used. The according device entry is created from the parameters and the entry is deleted. The IP address of the device is also removed from the network entry.
All public methods do not require the caller to specify the device of network entry, but use the private methods `_get_device_name` or `_get_network_name` to create those by using one or more of the parameters `ip`, `network` and `enabled`.
The method `get_device_mac` can be used to retrieve the hardware address of a specific device, whereas `get_devices` returns a list of IP address belonging to a specific network.
The `SelectiveDaemonApp` uses the `get_devices_values` method to retrieve a list containing a tuple for each device of a network. A tuple contains the IP address and the hardware address of a single device. In case a device does not exist, for example the caller asked for an enabled entry, although only a disabled one is stored, the Redis server returns the string `"None"` for this device. In avoidance of those values, the parameter `filter_values=True` can be provided to filter such values.
`get_pubsub` returns a `PubSub` object, which is used to subscribe to a specific channel and listen for messages.
In order to enable or disable specific device entries, the methods `disable_device` and `enable_device` were created.

Several private methods, which are beginning with an underscore, provide internal functionality and further abstraction. Those methods are available in appendix A.10.2.

As stated above, added device entries are removed by the Redis server after their TTL expires. The according IP address of the device in the network entry, is not automatically cleared by the system. The `PubSubThread`, located inside the module `lib/misc_thread.py` in appendix A.10.5, is used to listen for messages of the Redis keyspace event "expired" and deletes removed devices from the network entry. The following class-level variable is used to subscribe to the desired messages:

```
1 __SUBSCRIBE_TO = "__keyevent@{}__:expired"
2 """Used to subscribe to the keyspace event expired."""
```

The method `run` subscribes to the desired keyspace events of the database `ApateRedis.__DB` in line 2 and listens for messages afterwards. The `listen` method of the `redis PubSub` object, which is retrieved inside of the constructor with `get_pubsub` of the `ApateRedis` instance, blocks the execution of the thread until a new message is received. This messages contain the key of the deleted entry, that is used to remove the device from the network entry.

```
1 def run(self):
2     self.pubsub.subscribe(self.__SUBSCRIBE_TO.format(self.redis.__DB))
3     for message in self.pubsub.listen():
4         self.logger.debug("Removed expired device {} from network {}".format(util.
              get_device_ip(message['data']), util.get_device_net(message['data'])))
5         # removes the ip of the expired device (the removed device entry) from the
              network set
6         self.redis._del_device_from_network(util.get_device_ip(message['data']),
              util.get_device_net(message['data']))
```

4.6. Util

The module `lib/util.py` provides several methods used to facilitate often performed tasks, as listed below:

- `hex2str_mac(hex_val)` converts a hexadecimal MAC address inside a string into a human readable representation. For instance, if the broadcast address is passed in the format `"\xff\xff\xff\xff\xff\xff"`, the return value is `"ff:ff:ff:ff:ff:ff"`.
- `get_mac(ip, interface)` uses Scapy's `srp` method to retrieve the hardware address of the IP address `ip` using the network interface `interface`.

- `get_device_enabled(redis_device)` returns the enabled part of a device entry.
- `get_device_ip(redis_device)` returns the IP address part of a device entry.
- `get_device_net(redis_device)` returns the network address part of a device entry.

4.7. Changes to the Django Web Interface

The upribox web interface provides the possibility to change several configuration options of the upribox system in an intuitive way. This interface is extended by an option to enable or disable Apate inside the `/more` page. All files of the Django interface, that have been changed, are available in the appendix A.11.

First, a new URL used to toggle the status of Apate was added to the file `upribox_interface/urls.py`:

```
1 url(r'^more/apate/toggle$', "more.views.apate_toggle", name="upri_apate_toggle"),
```

This code defines, that the view `apate_toggle` is called if the site `more/apate/toggle` is accessed.

The new Django view `apate_toggle` is added to the file `upribox_interface/more/views.py`:

```
1 @login_required
2 def apate_toggle(request):
3     if request.method != 'POST':
4         raise Http404()
5
6     state = request.POST['enabled']
7     jobs.queue_job(sshjobs.toggle_apate, (state,))
8
9     return render_to_response("modal.html", {"message": True, "refresh_url": reverse
          ('upri_more')})
```

This function only allows the HTTP POST request method and uses the value of the parameter `enabled` in order to enable or disable the Apate daemon. The check of the value and the actions needed to change the state of Apate are performed by the `toggle_apate(state)` function inside the file `upribox_interface/more/jobs.py`. The view responds to the browser with a modal dialog, which is used to display the current status of the toggle process.

The function `toggle_apate` examines if the value of the parameter is `'yes'` or `'no'` as well as executes the script `/usr/local/bin/upri-config.py`, which applies changes via Ansible. In order to enable or disable Apate, the options `enable_apate` and `restart_apate` of the script are needed.

The following functions are the most important part of the `upri-config.py` script in regard to changing the state of the Apate daemon:

```
1  def action_set_apate(arg):
2      if arg not in ['yes', 'no']:
3          print 'error: only "yes" and "no" are allowed'
4          return 10
5      print 'apate enabled: %s' % arg
6      en = { "general": { "enabled": arg } }
7      write_role('apate', en)
8
9  def action_restart_apate(arg):
10     print 'restarting apate...'
11     return call_ansible('toggle_apate')
```

The first function `action_set_apate` is used to modify a local facts file for Apate, which determines the value of the variable `enabled` used during the Ansible deployment. The second function `action_restart_apate` executes the Ansible deployment with the tag "toggle_apate" and therefore causes the Apate daemon to be enabled or disabled on startup.
The template file `upribox_interface/more/templates/more.html` was extended by several lines, which displays the Apate section inside the web interface, provides a description as well as adds the switch to toggle the state of Apate. The source code can be seen in appendix A.11.4.

4.8. Test Environment

The implementation of the approach was constantly tested during the development. For this purpose, a separate network with access to the Internet was created. The Apate daemon was tested inside this network with several devices and different operating systems:

- a Linksys Wireless-G Broadband Router WRT54GL,
- an upribox using a Raspberry Pi 2 Model B v1.1 running Raspbian Jessie,
- a Raspberry Pi 1 Model B running Raspbian Wheezy,
- a Windows 10 device using a "Killer E2200 Gigabit Ethernet Controller" NIC[4] and an "Intel Dual Band Wireless-AC 7260" WNIC[5],

[4] Network Interface Controller
[5] Wireless Network Interface Controller

- an Ubuntu 14.10 device using a “RTL-8100/8101L/8139 PCI Fast Ethernet Adapter" NIC and also an “Intel Pro/Wireless 3945ABG" WNIC ,
- a Sony XPeria Z3 Compact running Android 6.0.1.

The upribox was used as the spoofing device by executing the Apate ARP spoofing daemon, which was deployed via Ansible. The other devices were used to test if the spoofing of the network was working and to simulate an average user’s Internet behaviour.
Due to the testing during the development of the holistic spoofing mode, several drawbacks of this mode of operation, as described in section 3.3.1, were discovered. This resulted in the development of the selective spoofing mode, which resolves the “unspoofing” problem, occuring because of the large amount of needed packets in order to spoof every possible host, by utilising a Redis database.

4.9. Results

During the tests of the Apate ARP spoofing daemon, no disadvantages regarding Internet and network activities of successfully spoofed devices compared to devices connected to the upribox wireless networks were detected.
In the course of tests using Apate’s holistic spoofing mode, some complications occurred. Rarely, the Windows 10 device using the “Killer E2200 Gigabit Ethernet Controller” NIC entered an inexplicable behaviour. While being successfully spoofed in the beginning, the network interface would start sending broadcast ARP requests asking for the gateway every few seconds, despite of receiving an answer of the upribox and the gateway every time. After processing an answer of the real gateway, the ARP tables entry was updated and the device was “unspoofed”. This result in a very instable networking state of the device, because of frequently dropping connections after being spoofed and “unspoofed” repeatedly. The behaviour would not stop after disabling the Apate daemon and restoring the normal network state and even continues after deactivating and reactivating the network interface. A reboot of the device is required in order to return the network interface into a normal way of funtioning.
This problem might be caused by a software bug of the network driver. Nevertheless, this occurrence shows that unpredictable issues could result in being unable to spoof a device or the whole network and could even interrupt the operations of the netwok. During the selective spoofing mode, which does not use GARP requests but unicast ARP replies to spoof devices, this behaviour did not occur.
While using a network address of `10.0.0.0` and a subnetmask of `255.0.0.0`, the holistic mode did not work as intended. The generation of packets for every possible host of the network takes a lot of time, due to the high number of possible clients. There is also a rather long delay every time the packets are

sent, which in fact results in not being able to spoof any device on the network and an extreme workload of the upribox.

Due to this test the selective spoofing mode, which utilises a database to store only information of existing hosts, became the preferred default mode of operation.

Results of testing the IGMP general query host discovery method show, that this method is only suited to be used supplementary. While the Windows device would reliably answer incoming queries, the Raspberry Pi 1 would not answer at all, due to not being a member of any multicast group. The Ubuntu 14.10 client did not respond to IGMP general queries, but only to group-specific queries. The Android device did not answer any queries, though it transmits IGMP group membership reports whenever the display of the device is enabled.

5. Discussion

This chapter illustrates the answer of the pivotal question, the criticism of the thesis and the implemented approach as well as future work.

5.1. Answering the Research Questions

The research question is: “Can ARP spoofing be a valid mechanism to defend privacy?”
As many other measures for increasing the Internet privacy or security of users, ARP spoofing can contribute to an enhanced privacy solution, but will not suffice if used alone. Just like a web filtering proxy is not able to increase privacy by filtering network traffic if no traffic is directed to it, ARP spoofing alone is not benefical by only redirecting traffic. Keeping the dependence of privacy enhancing features on other services in mind, ARP spoofing becomes a valid mechanism to defend privacy.

5.1.1. How is it possible to increase Internet privacy with the help of ARP spoofing?

Applying ARP spoofing inside a network will not contribute to an increased privacy, unless it is used in conjunction with other services, for example a web filtering proxy. By enhancing the upribox, an open-source project with the aim to provide an easy-to-use solution to raise Internet privacy with an ARP spoofing functionality, both main contributions are assembled. On the one hand the ARP spoofing solution redirects the traffic of network devices to the upribox, on the other hand the upribox is able to refine the received data.

5.1.2. How can a found approach be implemented?

The approach can be implemented as an additional Ansible role for the upribox project. Several changes to the existing roles need to be performed, as mention in section 4.2. The new Ansible role installs various dependencies as well as the actual ARP spoofing solution. This ARP spoofing daemon is written in Python and uses Scapy to perform several network tasks. The spoofing part of the daemon can be implemented using different techniques, for instance utilising GARP requests as done by the holistic

spoofing mode. Another technique only spoofs existing devices, which are stored inside a database, through ARP replies as done by the selective spoofing mode.

5.1.3. Which possibilities except ARP spoofing could be used for increasing the Internet privacy?

In reference to the increasing importance of IP version 6, the Neighbor Discovery protocol, which is used to perform address resolution similar to the Address Resolution Protocol for IP version 4, can be used to redirect network traffic. This possibility is further explained in section 5.3.2. Other techniques to redirect traffic from hosts to another target can also provide a sufficient solution, for instance ICMP redirects.

5.2. Criticism of the Thesis and the Implemented Approach

The environment used for testing the developed approach was artificially created for this purpose. It does not allow a comprehensive examination of every possibility regarding network devices, different network interface controllers, routers and operating systems. This could cause issues or other difficulties, which might occur in specific situations, to remain undetected. Another drawback due to the limited test environment is the fact, that the solution was not tested inside a “real world” network. The developer of the approach constructed the network and chose the devices and operating systems in reference to their understanding of an average network and user’s Internet behaviour. This perception might differ from several “real world” situations and result in unconsidered aspects.

The IGMP host discovery method, which was chosen as a supplementary technique for discovering existing devices, was implemented using IGMP general queries. As mentioned in section 4.9, several systems do not respond to IGMP general queries. The current implementation of this host discovery method does not consider this fact and does not take appropriate measures. The method should be extended by surveying appropriate multicast group addresses and adding functionality to query memberships of these specific groups. Other actions might be taken to further increase the benefit of this method.

Another aspect, which was not taken into account, is the existence of defense measures against ARP spoofing, as explained in section 2.3. In case the ARP messages are authenticated as proposed by S-ARP or TARP, the clients would not accept the forged packets of the upribox and therefore would not be spoofed. In networks using MR-ARP, clients, that receive a spoofed packet, would always approach the original gateway, also resulting in not being spoofed. Thus the developed approach is not applicable inside networks utilising such defence measures against ARP spoofing. In order to be able to integrate

this solution into protected networks further research and an adaption of the solution is needed.

5.3. Future Work

This chapter presents future work, which is needed in order to further improve the solution, besides the adaption of the test environment and the IGMP host discovery implementation as well as the needed research regarding ARP spoofing defense measures, as already mentioned in section 5.2.

5.3.1. Enhancing the Web Interface

Despite slightly contradicting the zero configuration ideology, more advanced users might be in need of an option to enable or disable the spoofing for specific devices. While the developed solution provides developers with an option to toggle the spoofing state of a device, the web interface lacks the functionality to exclude particular devices. The web interface only allows the complete deactivation of the Apate ARP spoofing daemon. Therefore the web interface should be extended by this feature according to the principles of usability and following the currently applied design.

5.3.2. Migration to IPv6

Due to the increasing importance of IPv6, the solution should be adapted to use methods appropriate for spoofing devices using IPv6. At present, the upribox does not provide configurations for the utilisation of IPv6 and thus does not support this protocol. By the time the upribox adds support for this protocol, the Apate ARP spoofing daemon should also be able to spoof IPv6 devices. As described in section 2.9, the address resolution part of the Neighbor Discovery Protocol functions in a very similar way to the Address Resolution Protocol. Therefore the spoofing could be performed by forging an unicast neighbor advertisement packet with following values for the most important fields:

- the hardware address of the network device as Ethernet destination,
- the hardware address of the upribox as Ethernet source,
- the value `0x86dd` representing IPv6 as Ethernet type,
- the IPv6 address of the network device as IPv6 destination,
- the IPv6 address of the upribox as IPv6 source,
- the value `1` to set the override flag,
- the IPV6 address of the upribox as ICMP target address,

- the hardware address of the upribox as ICMP target link-layer address.

Due to the present Override Flag, the receiving network device updates its existing neighbor cache entry of the gateway using the provided ICMP target link-layer address. The modified entry is set to the state STALE, because the client has to verify the reachability of the host. This behaviour could be avoided by additionally setting the solicited flag, which results in a state of REACHABLE for the updated entry.
As this method only works via unicast packets, further host discovery methods applicable for networks using IPv6 would be needed.

6. Conclusion

Concluding it can be said that the Apate ARP spoofing daemon, which is installed by an extension for the project upribox, can be used to successfully spoof all clients on the network and to redirect network traffic to the upribox, which in turn refines the received data in order to increase the user's Internet privacy.

The daemon provides two different modes of operation, which both spoof network hosts but have a distinct internal way of working. The holistic spoofing mode, described in the sections 3.3.1 and 4.4.3, requires less resources, but might be unsuited for bigger networks because of spoofing issues caused by the delay originating from the generation of packets for every possible host on the network. While using more resources, the selective spoofing mode, explained in the sections 3.3.2 and 4.4.4, resolves this drawback by only creating packets for existing hosts utilising a database and different host discovery techniques.

The author deems that despite the test environment being a limited and artificially created network, as mentioned in the sections 4.8 and 5.2, the Apate daemon is suited for the networks of a majority of users, because of the adaptability provided by the different modes of operation.

Especially the selective spoofing mode, which is chosen per default, resulted in the stable spoofing of all network devices. This mode is also preferable, because it only uses common ARP requests and replies, whereas the holistic spoofing mode makes use of GARP requests. As explained in section 4.9, during the tests some devices rarely showed strange behaviour after dealing with GARP requests. Such problems are less likely to happen with the selective spoofing mode, because common ARP packets are used a lot more frequent than GARP packets and therefore bugs in network drivers affecting the workflow of ARP are more likely to be detected and fixed.

Another positive aspect of the Apate ARP spoofing daemon is the contribution to the usability in accordance with the zero configuration ideology of the upribox project by automatically redirecting the network traffic to the upribox without requiring the user to change the configuration of the devices.

A. Source Code

A.1. tasks/main.yml

```
1  ---
2  - include: ../../common/tasks/other_env.yml
3  - include: apate_state.yml tags=toggle_apate
4
5  - name: create working directory for apate daemon
6    file: path=/opt/apate state=directory recurse=yes mode=0771 owner=root group=root
7
8  - name: copy the apate files
9    copy: src=apate/ dest=/opt/apate owner=root group=root mode=0774
10   notify: restart apate
11
12 - name: copy apate init script
13   template: src=init/apate dest=/etc/init.d/apate owner=root group=root mode=0755
14   notify: restart apate
15
16 - name: copy apate service file
17   template: src=init/apate.service dest=/etc/systemd/system/apate.service owner=root
         group=root mode=0755
18   notify: restart apate
19   register: service_file
20
21 - name: systemctl daemon-reload
22   shell: /bin/systemctl daemon-reload
23   when: service_file.changed
24
25 - name: create apate config dir
26   file: path=/etc/apate state=directory recurse=yes mode=0771 owner=root group=root
27
28 - name: copy apate config file
29   template: src=config.json dest=/etc/apate/config.json owner=root group=root mode
       =0755
```

```
30   notify: restart apate
31
32 - name: install virtualenv, tcpdump
33   apt: name={{ item }} state="{{ apt_target_state }}" force=yes update_cache=yes
         cache_valid_time="{{ apt_cache_time }}"
34   with_items:
35     - python-virtualenv
36     - tcpdump
37     - redis-server
38
39 - name: install requirements to virtualenv
40   pip: requirements=/opt/apate/requirements.txt virtualenv=/opt/apate/venv
41   notify: restart apate
42
43 - name: remove log files from other environment
44   file: path={{other_env.default_settings.log.general.path}}/{{other_env.
         default_settings.log.apate.subdir}} state=absent
45
46 - name: modify logrotate.d entry
47   template: src=logrotate.j2 dest=/etc/logrotate.d/apate mode=0644
48
49 - name: configure apate service
50   service: name=apate state='{{ "started" if apate_enabled|bool else "stopped" }}'
         enabled='{{ apate_enabled|bool }}'
51   tags:
52     - toggle_apate
53
54 - name: change keyspace event notification of redis-sever
55   lineinfile:
56     dest: /etc/redis/redis.conf
57     regexp: '^notify-keyspace-events'
58     line: 'notify-keyspace-events "Ex"'
59   notify: restart redis
60
61 - name: enable redis server
62   service: name=redis-server enabled=yes
```

A.2. tasks/apate_state

```
1 ---
```

```
2 #jinja2 has to evaluate this string seperately, because it is not possible to just
      include this string in a "when" statement
3 - set_fact:
4     apate_enabled: "{{ default_settings.apate.general.enabled if not (ansible_local
          is defined and ansible_local.apate is defined and ansible_local.apate.
          general is defined) else ansible_local.apate.general.enabled | default(
          default_settings.apate.general.enabled) }}"
```

A.3. handlers/main.yml

```
1 ---
2 - include: ../tasks/apate_state.yml tags=toggle_apate
3
4 - name: restart apate
5   service: name=apate state={{"restarted" if apate_enabled|bool else "stopped"}}
6
7 - name: restart redis
8   service: name=redis-server state=restarted
```

A.4. templates/apate

```
1  #! /bin/bash
2
3  ### BEGIN INIT INFO
4  # Provides: apate
5  # Required-Start: $local_fs $remote_fs $network $syslog $all
6  # Should-Start:
7  # Required-Stop: $local_fs $remote_fs $network $syslog
8  # Should-Stop:
9  # Default-Start:  2 3 4 5
10 # Default-Stop:   0 1 6
11 # Short-Description: ARP Spoofing Daemon
12 # Description:    Runs up the ARP Spoofing Daemon process
13 ### END INIT INFO
14
15 PATH=/sbin:/bin:/usr/sbin:/usr/bin:/usr/local/sbin:/usr/local/bin
16
17 if [[ $EUID -ne 0 ]]; then
18    echo "This daemon must be run as root"
```

```
19    exit 1
20 fi
21
22 # Activate the python virtual environment
23 . /opt/apate/venv/bin/activate
24
25 PID={{ default_settings.apate.pid.dir }}
26 LOGDIR={{ default_settings.log.general.path }}/{{ default_settings.log.apate.subdir
       }}
27
28 if [ ! -d $PID ]; then
29     mkdir $PID || return 2
30 fi
31
32 if [ ! -d $LOGDIR ]; then
33     mkdir $LOGDIR || return 2
34 fi
35
36 case "$1" in
37   start)
38     echo "Starting server"
39     # Start the daemon
40     python /opt/apate/apate.py start
41     ;;
42   stop)
43     echo "Stopping server"
44     # Stop the daemon
45     python /opt/apate/apate.py stop
46     ;;
47   restart)
48     echo "Restarting server"
49     python /opt/apate/apate.py restart
50     ;;
51   *)
52     # Refuse to do other stuff
53     echo "Usage: /etc/init.d/apate {start|stop|restart}"
54     exit 1
55     ;;
56 esac
57
58 exit 0
```

A.5. templates/apate.service

```
[Unit]
Description=Apate ARP Spoofing Daemon
Requires=networking.service network-online.target redis-server.service
Wants=network-online.target
After=networking.service redis-server.service network-online.target

[Service]
Type=forking
ExecStart=/etc/init.d/apate start
ExecStop=/etc/init.d/apate stop
PIDFile={{ default_settings.apate.pid.dir }}/{{ default_settings.apate.pid.file }}
Restart=on-failure

[Install]
WantedBy=multi-user.target
```

A.6. templates/logrotate.j2

```
{{default_settings.log.general.path}}/{{default_settings.log.apate.subdir}}/*.log {
    su root root
    compress
    copytruncate
    weekly
    rotate 3
    missingok
    notifempty
    maxsize 10M
}
```

A.7. templates/config.json

```
{
    "pidfile": "{{ default_settings.apate.pid.dir }}/{{ default_settings.apate.pid.
        file }}",
    "logfile": "{{ default_settings.log.general.path }}/{{ default_settings.log.
        apate.subdir }}/{{ default_settings.log.apate.logfiles.logname }}",
    "interface": "eth0",
```

```
5     "stdout": "{{ default_settings.log.general.path }}/{{ default_settings.log.apate
          .subdir }}/{{ default_settings.log.apate.logfiles.stdout }}",
6     "stderr": "{{ default_settings.log.general.path }}/{{ default_settings.log.apate
          .subdir }}/{{ default_settings.log.apate.logfiles.stderr }}",
7     "mode": "{{ default_settings.apate.mode }}"
8 }
```

A.8. environments/development/group_vars/all.yml

```
1  ---
2  env: development
3
4  remote_user: upri
5  sudo_group: upriusers
6  hostname: upribox
7  remote_user_login_shell: /bin/bash
8
9  apt_cache_time: 3600
10 apt_target_state: installed
11
12 ansible_pip_version: 1.9.6
13 ansible_uwsgi_version: 1.9.2
14
15 django_settings_file: settings_dev
16
17 default_settings:
18   apate:
19     general:
20       enabled: 'yes'
21     # modes: selective, holistic
22     mode: 'selective'
23     pid:
24       dir: '/var/run/apate'
25       file: 'apate.pid'
26   django:
27     # this path is for "upri-config.py parse_logs"
28     # if you want to change the path, you will also have to
29     # edit django's settings file
30     db: '/usr/share/nginx/www-upri-interface/db.sqlite3'
31   tor:
```

```
32      general:
33        enabled: 'no'
34    ssh:
35      general:
36        enabled: 'yes'
37    vpn:
38      general:
39        enabled: 'yes'
40    log:
41      general:
42        path: '/var/log/log'
43      privoxy:
44        subdir: 'privoxy'
45        logfiles:
46          logname: 'privoxy.log'
47      apate:
48        subdir: 'apate'
49        logfiles:
50          logname: apate.log
51          stdout: stdout.log
52          stderr: stderr.log
53      tor:
54        subdir: 'tor'
55        logfiles:
56          logname:  'log'
57      rqworker:
58        logfiles:
59          stdout: 'rqworker.out.log'
60          stderr: 'rqworker.err.log'
61      supervisor:
62        logfiles:
63          logname: 'supervisord.log'
64      uwsgi:
65        logfiles:
66          logname: 'uwsgi.log'
67      dnsmasq:
68        logfiles:
69          logname: 'dnsmasq.log'
70      dnsmasq_ninja:
71        logfiles:
72          logname: 'dnsmasq_ninja.log'
```

```
73      nginx:
74        logfiles:
75          error: 'nginx_error.log'
76          access: 'nginx_access.log'
77          interface_error: 'nginx_interface_error.log'
78          interface_access: 'nginx_interface_access.log'
79          blackhole_error: 'nginx_blackhole_error.log'
80          blackhole_access: 'nginx_blackhole_access.log'
81          css_error: 'nginx_css_error.log'
82          css_access: 'nginx_css_access.log'
83      vpn:
84        logfiles:
85          logname: 'openvpn.log'
86          status: 'openvpn-status.log'
87      rsyslog:
88        subdir: 'rsyslog'
89        logfiles:
90            auth: 'auth.log'
91            syslog: 'syslog'
92            cron: 'cron.log'
93            daemon: 'daemon.log'
94            kern: 'kern.log'
95            lpr: 'lpr.log'
96            mail: 'mail.log'
97            user: 'user.log'
98            mail_info: 'mail.info'
99            mail_warn: 'mail.warn'
100           mail_error: 'mail.error'
101           news_crit: 'news.crit'
102           news_err: 'news.err'
103           news_notice: 'news.notice'
104           debug: 'debug'
105           messages: 'messages'
106
107 # variables for pull updates in development mode
108 pull_cron_schedule: '0 */4 * * *'
109 pull_cron_user: root
110 pull_logfile: /var/log/ansible-pull.log
111 pull_workdir: /var/lib/ansible/local
112 pull_branch: master
113 pull_repo_url: git@github.com:usableprivacy/upribox.git
```

```
114 pull_git_host: github.com
115 pull_git_sshkey: '|1|YSIl0HnC//DkFvWLLsAsBxDU10Q=|PC/XJu88KyBvcN7nilLAbany2bE= ssh-
        rsa AAAAB3NzaC1yc2EAAAABIwAAAQEAq2A7hRGmdnm9tUDbO9IDSwBK6TbQa+
        PXYPCPy6rbTrTtw7PHkccKrpp0yVhp5HdEIcKr6pLlVDBfOLX9QUsyCOV0wzfjIJNlGEYsdlLJizHhbn2mUjvSAHQqZ
        /yMf+Se8xhHTvKSCZIFImWwoG6mbUoWf9nzpIoaSjB+weqqUUmpaaasXVal72J+UX2B+2
        RPW3RcT0eOzQgqlJL3RKrTJvdsjE3JEAvGq3lGHSZXy28G3skua2SmVi/w4yCE6gbODqnTWlg7+
        wC604ydGXA8VJiS5ap43JXiUFFAaQ=='
```

A.9. environments/production/group_vars/all.yml

```
1  ---
2  env: production
3
4  remote_user: upri
5  sudo_group: upriusers
6  hostname: upribox
7  remote_user_login_shell: /bin/bash
8  apt_cache_time: 86400
9  apt_target_state: installed
10
11 # variables for pull updates in production mode
12 pull_cron_schedule: '0 */4 * * *'
13 pull_cron_user: root
14 #pull_logfile: /var/log/ansible-pull.log
15 pull_workdir: /var/lib/ansible/local
16 pull_branch: master
17 pull_repo_url: git@github.com:usableprivacy/upribox.git
18 pull_git_host: github.com
19 pull_git_sshkey: '|1|YSIl0HnC//DkFvWLLsAsBxDU10Q=|PC/XJu88KyBvcN7nilLAbany2bE= ssh-
       rsa AAAAB3NzaC1yc2EAAAABIwAAAQEAq2A7hRGmdnm9tUDbO9IDSwBK6TbQa+
       PXYPCPy6rbTrTtw7PHkccKrpp0yVhp5HdEIcKr6pLlVDBfOLX9QUsyCOV0wzfjIJNlGEYsdlLJizHhbn2mUjvSAHQqZ
       /yMf+Se8xhHTvKSCZIFImWwoG6mbUoWf9nzpIoaSjB+weqqUUmpaaasXVal72J+UX2B+2
       RPW3RcT0eOzQgqlJL3RKrTJvdsjE3JEAvGq3lGHSZXy28G3skua2SmVi/w4yCE6gbODqnTWlg7+
       wC604ydGXA8VJiS5ap43JXiUFFAaQ=='
20
21 ansible_pip_version: 1.9.6
22 ansible_uwsgi_version: 1.9.2
23
24 django_settings_file: settings_prod
25
```

```
default_settings:
  apate:
    general:
      enabled: 'yes'
    # modes: selective, holistic
    mode: 'selective'
    pid:
      dir: '/var/run/apate'
      file: 'apate.pid'
  django:
    # this path is for "upri-config.py parse_logs"
    # if you want to change the path, you will also have to
    # edit django's settings file
    db: '/var/upribox-interface/db.sqlite3'
  tor:
    general:
      enabled: 'no'
  ssh:
    general:
      enabled: 'yes'
  vpn:
    general:
      enabled: 'no'
  log:
    general:
      path: '/var/tmp/log'
    ansible_pull:
      logfiles:
        logname: 'ansible-pull.log'
    privoxy:
      subdir: 'privoxy'
      logfiles:
        #logrotate applies to *.log
        logname: 'privoxy.log'
    apate:
      subdir: 'apate'
      logfiles:
        logname: apate.log
        stdout: stdout.log
        stderr: stderr.log
    tor:
```

```
67        subdir: 'tor'
68        logfiles:
69          logname:  'log'
70      rqworker:
71        logfiles:
72          stdout: 'rqworker.out.log'
73          stderr: 'rqworker.err.log'
74      supervisor:
75        logfiles:
76          logname: 'supervisord.log'
77      uwsgi:
78        logfiles:
79          logname: 'uwsgi.log'
80      dnsmasq:
81        logfiles:
82          logname: 'dnsmasq.log'
83      dnsmasq_ninja:
84        logfiles:
85          logname: 'dnsmasq_ninja.log'
86      nginx:
87        logfiles:
88          error: 'nginx_error.log'
89          access: 'nginx_access.log'
90          interface_error: 'nginx_interface_error.log'
91          interface_access: 'nginx_interface_access.log'
92          blackhole_error: 'nginx_blackhole_error.log'
93          blackhole_access: 'nginx_blackhole_access.log'
94          css_error: 'nginx_css_error.log'
95          css_access: 'nginx_css_access.log'
96      vpn:
97        logfiles:
98          logname: 'openvpn.log'
99          status: 'openvpn-status.log'
100     rsyslog:
101       subdir: 'rsyslog'
102       logfiles:
103           auth: 'auth.log'
104           syslog: 'syslog'
105           cron: 'cron.log'
106           daemon: 'daemon.log'
107           kern: 'kern.log'
```

```
108             lpr: 'lpr.log'
109             mail: 'mail.log'
110             user: 'user.log'
111             mail_info: 'mail.info'
112             mail_warn: 'mail.warn'
113             mail_error: 'mail.error'
114             news_crit: 'news.crit'
115             news_err: 'news.err'
116             news_notice: 'news.notice'
117             debug: 'debug'
118             messages: 'messages'
```

A.10. files/apate

A.10.1. apate.py

```
1  #!/usr/bin/env python
2  # coding=utf-8
3  """This script is used to control the Apate ARP spoofing daemon."""
4  import logging
5  import sys
6  import signal
7  import os
8  import json
9  import lockfile
10
11 from daemon import runner
12
13 from lib import daemon_app
14
15 CONFIG_FILE = "/etc/apate/config.json"
16 """Path of the config file for the Apate ARP spoofing daemon."""
17 CONFIG_OPTIONS = ('logfile', 'pidfile', 'interface', 'stderr', 'stdout', 'mode')
18 """Options that need to be present in the config file."""
19
20
21 def main():
22     """This script is used to initialise and command the Apate ARP spoofing daemon.
23     It parses the configuration file at CONFIG_FILE and checks for the necessary
           Options
```

```
24      CONFIG_OPTIONS. The daemon needs to be run as root.
25      """
26      # check if run as root
27      if os.geteuid() != 0:
28          print "This daemon needs to be run as root"
29          sys.exit(1)
30
31      # parse configuration file
32      try:
33          with open(CONFIG_FILE) as config:
34              data = json.load(config)
35      except ValueError as ve:
36          print "Could not parse the configuration file"
37          print str(ve)
38          sys.exit(3)
39      except IOError as ioe:
40          print "An error occurred while trying to open the configuration file"
41          print str(ioe)
42          sys.exit(4)
43
44      # check if all necessary options are present in config file
45      if not all(val in data for val in CONFIG_OPTIONS):
46          print "The configuration file does not include all necessary options"
47          sys.exit(2)
48
49      # set up logger for daemon
50      logger = logging.getLogger("DaemonLog")
51      logger.setLevel(logging.INFO)
52      formatter = logging.Formatter("%(asctime)s - %(name)s - %(levelname)s - %(
            message)s")
53      handler = logging.FileHandler(data['logfile'])
54      handler.setFormatter(formatter)
55      logger.addHandler(handler)
56
57      # catch error which could arise during initialisation
58      try:
59          # when configured to use the holistic spoofing mode
60          if data['mode'] == "holistic":
61              dapp = daemon_app.HolisticDaemonApp(logger, str(data['interface']), data
                    ['pidfile'], data['stdout'], data['stderr'])
62          else:
```

```
63              # selective spoofing mode is default
64              dapp = daemon_app.SelectiveDaemonApp(logger, str(data['interface']),
                    data['pidfile'], data['stdout'], data['stderr'])
65      except Exception as e:
66          logger.error("An error happened during initialsising the daemon process -
                terminating process")
67          logger.exception(e)
68          sys.exit(1)
69
70      # intialise daemon
71      daemon_runner = runner.DaemonRunner(dapp)
72      # don't close logfile
73      daemon_runner.daemon_context.files_preserve = [handler.stream]
74      # start cleanup routine when stopping the daemon
75      daemon_runner.daemon_context.signal_map[signal.SIGTERM] = dapp.exit
76
77      # command daemon
78      try:
79          daemon_runner.do_action()
80      except runner.DaemonRunnerError as dre:
81          print str(dre)
82      except lockfile.LockTimeout as lt:
83          # runner only catches AlreadyLocked, which is not thrown if a timeout was
                specified other than None or 0
84          # Following is thrown otherwise and slips through:
85          # LockTimeout: Timeout waiting to acquire lock for /var/run/apate/apate.pid
86          # though this should not be logged as an exception
87
88          # restart fails if timeout is set to 0 or None
89          print str(lt)
90      except Exception as e:
91          # log stacktrace of exceptions that should not occur to logfile
92          logger.error("Exception at do_action()")
93          logger.exception(e)
94
95
96  if __name__ == "__main__":
97      main()
```

A.10.2. lib/apate_redis.py

```
1  # coding=utf-8
2  """This module provides the ApateRedis class for managing the Apate Redis DB."""
3  import redis
4
5
6  class ApateRedis(object):
7      """This class is used to manage the Apate Redis DB."""
8      __PREFIX = "apate"
9      """str: Prefix which is used for every key in the redis db."""
10     __DELIMITER = ":"
11     """str: Delimiter used for separating parts of keys in the redis db."""
12     __IP = "ip"
13     """str: Indicator for the IP part of the redis key."""
14     __NETWORK = "net"
15     """str: Indicator for the network address part of the redis key."""
16     __DB = 5
17     """int: Redis db which should be used."""
18     __TTL = 259200
19     """int: Time after which device entries in the redis db expire. (default=3 days)
           """
20
21     def __init__(self, network, logger):
22         """Initialises ApateRedis objects. A connection to the local redis server
23         on port 6379 using the database, which is specified by __DB, is established.
24
25         Args:
26             network (str): Network IP Address, which should be used per default.
27             logger (logging.Logger): Used for logging messages.
28
29         """
30         self.redis = redis.StrictRedis(host="localhost", port=6379, db=self.__DB)
31         self.network = network
32         self.logger = logger
33
34     def add_device(self, ip, mac, network=None, enabled=True, force=False):
35         """Adds a device entry to the redis db. Checks if a disabled entry exists,
36         if not adds the device to the network redis set and adds a device entry.
37         Added devices are automatically removed after __TTL expires.
38
39         Args:
```

```
40              ip (str): IP address of the device.
41              mac (str): MAC (layer 2) address of the device.
42              network (str, optional): Network address of the device. If not set, self
                    .network is used instead.
43              enabled (bool, optional): Determines if the device should be enabled or
                    disabled. Defaults to True.
44              force (bool, optional): Switch used to force the insertion of the device
                     entry (even if a disabled entry already exists).
45                  Defaults to False.
46
47          Return:
48              bool: True if successful, false otherwise.
49
50          """
51          if not self._check_device_disabled(ip, network or self.network) or force:
52              self._add_device_to_network(ip, network or self.network)
53              return self._add_entry(self._get_device_name(ip, network or self.network
                    , enabled=enabled), mac)
54
55      def remove_device(self, ip, network=None, enabled=True):
56          """Removes a device from the network redis set and deletes the device entry.
57
58          Args:
59              ip (str): IP address of the device.
60              network (str, optional): Network address of the device. If not set, self
                    .network is used instead.
61              enabled (bool, optional): Determines if the device is enabled or
                    disabled. Defaults to True.
62
63          Results:
64              int: Number of devices deleted.
65
66          """
67          self._del_device_from_network(ip, network or self.network)
68          return self._del_device(self._get_device_name(ip, network or self.network,
                enabled=enabled))
69
70      def get_device_mac(self, ip, network=None, enabled=True):
71          """Returns the mac address (value from the redis db) of the device.
72
73          Args:
```

```
74              ip (str): IP address of the device.
75              network (str, optional): Network address of the device. If not set, self
                    .network is used instead.
76              enabled (bool, optional): Determines if the device is enabled or
                    disabled. Defaults to True.
77
78          Returns:
79              The mac address of the specified device as str if successful, None
                    otherwise.
80
81          """
82          return self.redis.get(self._get_device_name(ip, network or self.network,
                enabled=enabled))
83
84      def get_devices(self, network=None):
85          """Returns a list with ip addresses of devices of the specified network.
86
87          Args:
88              network (str, optional): Network address of the device. If not set, self
                    .network is used instead.
89
90          Returns:
91              List containing device ip addresses if successful, None otherwise.
92
93          """
94          return self.redis.smembers(self._get_network_name(network or self.network))
95
96      def get_devices_values(self, filter_values=False, network=None, enabled=True):
97          """Returns a list with tuples containing ip addresses of devices and the mac
                addresses of
98          the devices.
99
100         Args:
101             filter_values (bool, optional): Determines if str(None) values should be
                    filtered. Defaults to False.
102             network (str, optional): Network address of the device. If not set, self
                    .network is used instead.
103             enabled (bool, optional): Determines if the device is enabled or
                    disabled. Defaults to True.
104
105         Returns:
```

```
106             list: List with tuples containing ip addresses of devices and the mac
                    addresses of
107             the devices. May contain str(None) values, if there is no device entry
108             for the according ip address and filter_values = False.
109             E.g.:
110             {"192.168.0.1": "11:22:33:44:55:66", "192.168.0.2": "None"}
111
112         """
113         # list may contain null values
114         devs = self.get_devices(network=network or self.network)
115         if not devs:
116             return []
117         else:
118             vals = self.redis.mget([self._get_device_name(dev, network or self.
                    network, enabled=enabled) for dev in devs])
119             res = zip(devs, vals)
120             # filter "None" entries if filter_values is True
121             return res if not filter_values else [x for x in res if x[1] and x[1] !=
                     str(None)]  # filter(None, res)
122
123     def get_pubsub(self, ignore_subscribe_messages=True):
124         """Used to get a PubSub object.
125
126         Args:
127             ignore_subscribe_messages (bool, optional): Determines if subscriptions
                    messages should be ignored. Defaults to True.
128
129         Returns:
130             redis.PubSub: PubSub object, which can be used to subscribe to redis
                    messages.
131
132         """
133         return self.redis.pubsub(ignore_subscribe_messages=ignore_subscribe_messages
                )
134
135     def disable_device(self, ip, network=None):
136         """Disables an enabled device in the redis db.
137         An enabled entry "apate:net:192.168.0.0:ip:192.168.0.1:1"
138         afterwards looks like this "apate:net:192.168.0.0:ip:192.168.0.1:0"
139
140         Args:
```

```
141             ip (str): IP address of the device.
142             network (str, optional): Network address of the device. If not set, self
                    .network is used instead.
143
144         """
145         self._toggle_device(ip, network or self.network, enabled=False)
146
147     def enable_device(self, ip, network=None):
148         """Enables a disabled device in the redis db.
149         An enabled entry "apate:net:192.168.0.0:ip:192.168.0.1:0"
150         afterwards looks like this "apate:net:192.168.0.0:ip:192.168.0.1:1"
151
152         Args:
153             ip (str): IP address of the device.
154             network (str, optional): Network address of the device. If not set, self
                    .network is used instead.
155
156         """
157         self._toggle_device(ip, network or self.network, enabled=True)
158
159     def _add_entry(self, key, value):
160         # inserted keys expire after __TTL
161         return self.redis.set(key, value, ApateRedis.__TTL)
162
163     def _del_device(self, device):
164         return self.redis.delete(device)
165
166     @staticmethod
167     def _get_device_name(ip, network, enabled=None):
168         # example for the return value
169         # ip = "192.168.0.1", network = "192.168.0.0", enabled = True  -->  "apate:
                net:192.168.0.0:ip:192.168.0.1:1"
170         if enabled is None:
171             # don't include the enabled-section (e.g.: "apate:net:192.168.0.0:ip
                    :192.168.0.1")
172             return ApateRedis.__DELIMITER.join((ApateRedis.__PREFIX, ApateRedis.
                    __NETWORK, str(network), ApateRedis.__IP, str(ip)))
173         else:
174             return ApateRedis.__DELIMITER.join((ApateRedis.__PREFIX, ApateRedis.
                    __NETWORK, str(network), ApateRedis.__IP, str(ip), str(int(enabled))
                    ))
```

```
175
176     @staticmethod
177     def _get_network_name(network):
178         # e.g.: network = "192.168.0.0"  -->  "apate:net:192.168.0.0"
179         return ApateRedis.__DELIMITER.join((ApateRedis.__PREFIX, ApateRedis.
                __NETWORK, str(network)))
180
181     def _add_device_to_network(self, ip, network):
182         """Adds an IP address to a network (redis set)."""
183         return self.redis.sadd(ApateRedis.__DELIMITER.join((ApateRedis.__PREFIX,
                ApateRedis.__NETWORK, str(network))), str(ip))
184
185     def _del_device_from_network(self, ip, network):
186         """Removes an IP address from a network (redis set)."""
187         return self.redis.srem(ApateRedis.__DELIMITER.join((ApateRedis.__PREFIX,
                ApateRedis.__NETWORK, str(network))), str(ip))
188
189     def _check_device_disabled(self, ip, network):
190         # True if devices is disabled
191         return self.redis.get(self._get_device_name(ip, network, enabled=False)) is
                not None
192
193     def _toggle_device(self, ip, network, enabled):
194         # add new device first and delete old device afterwards
195         # this is done to avoid race conditions
196         self.add_device(ip, self.get_device_mac(ip, network, enabled=not enabled),
                network, enabled=enabled, force=True)
197         self.remove_device(ip, network, enabled=not enabled)
```

A.10.3. lib/daemon_app.py

```
1  # coding=utf-8
2  """This module provides several classes that are used to implement a ARP spoofing
       daemon.
3
4  Classes:
5      _DaemonApp: Abstract class, that should be inherited.
6      HolisticDaemonApp: Inherits _DaemonApp and implements the holistist spoofing
           mode.
7      SelectiveDaemonApp: Inherits _DaemonApp and implements the selective spoofing
           mode.
```

```
8       DaemonError: Error that indicates the daemon's failure.
9
10  """
11  import os
12  import logging
13  import time
14  import netifaces as ni
15  from netaddr import IPAddress, IPNetwork, AddrFormatError
16
17  logging.getLogger("scapy.runtime").setLevel(logging.ERROR)
18  # suppresses following message
19  # WARNING: No route found for IPv6 destination :: (no default route?)
20  from scapy.all import conf, sendp, ARP, Ether, ETHER_BROADCAST
21
22  import util
23  from sniff_thread import HolisticSniffThread, SelectiveSniffThread
24  from apate_redis import ApateRedis
25  from misc_thread import ARPDiscoveryThread, IGMPDiscoveryThread, PubSubThread
26
27
28  class _DaemonApp(object):
29      """This is an abstract class, which should be inherited to define the
30      Apate daemon's behaviour."""
31
32      def __init__(self, logger, interface, pidfile, stdout, stderr):
33          """Initialises several things needed to define the daemons behaviour.
34
35          Args:
36              logger (logging.Logger): Used for logging messages.
37              interface (str): The network interface which should be used. (e.g. eth0)
38              pidfile (str): Path of the pidfile, used by the daemon.
39              stdout (str): Path of stdout, used by the daemon.
40              stderr (str): Path of stderr, used by the daemon.
41
42          Raises:
43              DaemonError: Signalises the failure of the daemon.
44          """
45          # disable scapys verbosity global
46          conf.verb = 0
47
48          self.stdin_path = os.devnull
```

```
49          self.stdout_path = stdout
50          self.stderr_path = stderr
51          self.pidfile_path = pidfile
52          self.pidfile_timeout = 5
53          # self.pidfile_timeout = 0
54
55          self.logger = logger
56          self.interface = interface
57
58          if_info = None
59          try:
60              if_info = ni.ifaddresses(self.interface)
61          except ValueError as e:
62              self.logger.error("An error concerning the interface {} has occurred: {}
                    ".format(self.interface, str(e)))
63              raise DaemonError()
64
65          # get ip of specified interface
66          self.ip = if_info[2][0]['addr']
67          # get subnetmask of specified interface
68          self.netmask = if_info[2][0]['netmask']
69          # get mac address of specified interface
70          self.mac = if_info[17][0]['addr']
71
72          # get network address
73          try:
74              self.network = IPNetwork("{}/{}".format(self.ip, self.netmask))
75          except AddrFormatError as afe:
76              # this should never happen, because values are retrieved via netifaces
                    library
77              self.logger.error("A grave error happened during determinig the network:
                     {}".format(str(afe)))
78              raise DaemonError()
79
80          # get default gateway
81          try:
82              self.gateway = ni.gateways()["default"][ni.AF_INET][0]
83          except KeyError:
84              self.logger.error("No default gateway is configured")
85              raise DaemonError()
86
```

```
87          # get all ip addresses that are in the specified network
88          # and remove network address, broadcast, own ip, gateway ip
89          self.ip_range = list(self.network)
90          self.ip_range.remove(IPAddress(self.ip))
91          self.ip_range.remove(IPAddress(self.gateway))
92          self.ip_range.remove(IPAddress(self.network.broadcast))
93          self.ip_range.remove(IPAddress(self.network.network))
94
95          try:
96              # get MAC address of gateway
97              self.gate_mac = util.get_mac(self.gateway, self.interface)
98              if not self.gate_mac:
99                  raise DaemonError()
100         except Exception:
101             self.logger.error("Unable to get MAC address of Gateway")
102             raise DaemonError()
103
104     def _return_to_normal(self):
105         """This method should be overriden to define the actions to be done when
                stopping the daemon."""
106         pass
107
108     def exit(self, signal_number, stack_frame):
109         """This method is called if the daemon stops."""
110         self._return_to_normal()
111         raise SystemExit()
112
113     def run(self):
114         """This method should be overriden to define the daemon's behaviour."""
115         pass
116
117
118 class HolisticDaemonApp(_DaemonApp):
119     """Implements the abstract class _DaemonApp and also implements the holistic
            spoofing mode of Apate.
120     The holistic spoofing mode requires less resources than the selective spoofing
            mode,
121     e.g.: redis-server is not needed. This mode is suitable for small networks (e.g.
             /24).
122     """
123
```

```
124     __SLEEP = 20
125     """int: Defines the time to sleep after packets are sent before they are sent
            anew."""
126
127     def __init__(self, logger, interface, pidfile, stdout, stderr):
128         """Initialises several things needed to define the daemons behaviour.
129
130         Args:
131             logger (logging.Logger): Used for logging messages.
132             interface (str): The network interface which should be used. (e.g. eth0)
133             pidfile (str): Path of the pidfile, used by the daemon.
134             stdout (str): Path of stdout, used by the daemon.
135             stderr (str): Path of stderr, used by the daemon.
136
137         Raises:
138             DaemonError: Signalises the failure of the daemon.
139         """
140         super(self.__class__, self).__init__(logger, interface, pidfile, stdout,
                stderr)
141
142         self.sniffthread = HolisticSniffThread(self.interface, self.gateway, self.
                mac, self.gate_mac)
143         self.sniffthread.daemon = True
144
145     def _return_to_normal(self):
146         """This method is called when the daemon is stopping.
147         First, sends a GARP broadcast request to all clients to tell them the real
                gateway.
148         Then an ARP request is sent to every client, so that they answer the real
                gateway and update its ARP cache.
149         """
150         # clients gratutious arp
151         sendp(
152             Ether(dst=ETHER_BROADCAST) / ARP(op=1, psrc=self.gateway, pdst=self.
                    gateway, hwdst=ETHER_BROADCAST,
153                                              hwsrc=self.gate_mac))
154         # to clients so that they send and arp reply to the gateway
155         sendp(Ether(dst=ETHER_BROADCAST) / ARP(op=1, psrc=self.gateway, pdst=str(
                self.network), hwsrc=self.gate_mac))
156
157     def exit(self, signal_number, stack_frame):
```

```
158         """This method is called from the python-daemon when the daemon is stopping.
159         Threads are stopped and clients are despoofed via _return_to_normal().
160         """
161         self._return_to_normal()
162         raise SystemExit()
163 
164     def run(self):
165         """Starts the thread, which is sniffing incoming ARP packets and sends out
                packets to spoof
166         all clients on the network and the gateway. This packets are sent every
                __SLEEP seconds.
167 
168         Note:
169             First, a ARP request packet is generated for every possible client of
                    the network.
170             This packets are directed at the gateway and update existing entries of
                    the gateway's ARP table.
171             So the gateway is not flooded with entries for non-existing clients.
172 
173             Second, a GARP broadcast request packet is generated to spoof every
                    client on the network.
174         """
175         # start sniffing thread
176         self.sniffthread.start()
177 
178         # generates a packet for each possible client of the network
179         # these packets update existing entries in the arp table of the gateway
180         packets = [Ether(dst=self.gate_mac) / ARP(op=1, psrc=str(x), pdst=str(x))
                for x in self.ip_range]
181         # gratuitous arp to clients
182         # updates the gateway entry of the clients arp table
183         packets.append(Ether(dst=ETHER_BROADCAST) / ARP(op=1, psrc=self.gateway,
                pdst=self.gateway,
184                                                         hwdst=ETHER_BROADCAST))
185         while True:
186             sendp(packets)
187             time.sleep(self.__SLEEP)
188 
189 
190 class SelectiveDaemonApp(_DaemonApp):
```

```
191     """Implements the abstract class _DaemonApp and also implements the selective
            spoofing mode of Apate.
192     The selective spoofing mode requires more resources than the holistic spoofing
            mode,
193     e.g.: the redis-server. This mode only generates packets for existing clients (
            not every possible client).
194     This mode is suitable for bigger networks, as the bottleneck of this mode is
            virtually only the host discovery.
195     """
196
197     __SLEEP = 5
198     """int: Defines the time to sleep after packets are sent before they are sent
            anew."""
199
200     def __init__(self, logger, interface, pidfile, stdout, stderr):
201         """Initialises several things needed to define the daemons behaviour.
202
203         Args:
204             logger (logging.Logger): Used for logging messages.
205             interface (str): The network interface which should be used. (e.g. eth0)
206             pidfile (str): Path of the pidfile, used by the daemon.
207             stdout (str): Path of stdout, used by the daemon.
208             stderr (str): Path of stderr, used by the daemon.
209
210         Raises:
211             DaemonError: Signalises the failure of the daemon.
212         """
213         super(self.__class__, self).__init__(logger, interface, pidfile, stdout,
                stderr)
214         self.redis = ApateRedis(str(self.network.network), logger)
215
216         # Initialise threads
217         self.sniffthread = SelectiveSniffThread(self.interface, self.gateway, self.
                mac, self.gate_mac, self.redis)
218         self.sniffthread.daemon = True
219         self.psthread = PubSubThread(self.redis, self.logger)
220         self.psthread.daemon = True
221         self.arpthread = ARPDiscoveryThread(self.gateway, str(self.network.network))
222         self.arpthread.daemon = True
223         self.igmpthread = IGMPDiscoveryThread(self.gateway, str(self.network.network
                ), self.ip, self.mac)
```

```
224         self.igmpthread.daemon = True
225
226     def _return_to_normal(self):
227         """This method is called when the daemon is stopping.
228         First, sends a GARP broadcast request to all clients to tell them the real
                gateway.
229         Then ARP replies for existing clients are sent to the gateway.
230         """
231         # spoof clients with GARP boradcast request
232         sendp(
233             Ether(dst=ETHER_BROADCAST) / ARP(op=1, psrc=self.gateway, pdst=self.
                    gateway, hwdst=ETHER_BROADCAST,
234                                              hwsrc=self.gate_mac))
235
236         # generate ARP reply packet for every existing client and spoof the gateway
237         packets = [Ether(dst=self.gate_mac) / ARP(op=2, psrc=dev[0], pdst=self.
                gateway, hwsrc=dev[1]) for dev in self.redis.get_devices_values(
                filter_values=True)]
238         sendp(packets)
239
240     def exit(self, signal_number, stack_frame):
241         """This method is called from the python-daemon when the daemon is stopping.
242         Threads are stopped and clients are despoofed via _return_to_normal().
243         """
244         self._return_to_normal()
245         raise SystemExit()
246
247     def run(self):
248         """Starts multiple threads sends out packets to spoof
249         all existing clients on the network and the gateway. This packets are sent
                every __SLEEP seconds.
250         The existing clients (device entries) are read from the redis database.
251
252         Threads:
253             A SniffThread, which sniffs for incoming ARP packets and adds new
                    devices to the redis db.
254             Two HostDiscoveryThread, which are searching for existing devices on the
                     network.
255             A PubSubThread, which is listening for redis expiry messages.
256
257         Note:
```

```
258             First, ARP replies to spoof the gateway entry of existing clients arp
                    cache are generated.
259             ARP relpies to spoof the entries of the gateway are generated next.
260             Unlike the holistic mode only packets for existing clients are generated
                    .
261
262         """
263         self.sniffthread.start()
264         self.arpthread.start()
265         self.psthread.start()
266         self.igmpthread.start()
267
268         # lamda expression to generate arp replies to spoof the clients
269         exp1 = lambda dev: Ether(dst=dev[1]) / ARP(op=2, psrc=self.gateway, pdst=dev
                [0], hwdst=dev[1])
270
271         # lamda expression to generate arp replies to spoof the gateway
272         exp2 = lambda dev: Ether(dst=self.gate_mac) / ARP(op=2, psrc=dev[0], pdst=
                self.gateway, hwdst=self.gate_mac)
273
274         while True:
275             # generates packets for existing clients
276             # due to the labda expressions p1 and p2 this list comprehension, each
                    iteration generates 2 packets
277             # one to spoof the client and one to spoof the gateway
278             packets = [p(dev) for dev in self.redis.get_devices_values(filter_values
                    =True) for p in (exp1, exp2)]
279
280             sendp(packets)
281             time.sleep(self.__SLEEP)
282
283
284 class DaemonError(Exception):
285     """This error class indicates, that the daemon has failed."""
286     pass
```

A.10.4. lib/extended_runner.py

```
1 # coding=utf-8
2 """This module provides the class ExtendedRunner, which can be used to perfom
3 additional actions with the daemon context."""
```

```
4  from daemon import runner
5
6
7  class ExtendedRunner(runner.DaemonRunner):
8      """This class can be used to perfom additional actions with the daemon context.
           """
9
10     def __init__(self, app):
11         super(self.__class__, self).__init__(app)
12         # extend __init__ here
13         # do something additional with daemon context
```

A.10.5. lib/misc_thread.py

```
1  # coding=utf-8
2  """This module provides several threads used by the ARP spoofing daemon.
3
4  Classes:
5      ARPDiscoveryThread: Discovers clients on the network by sending out ARP request.
6      IGMPDiscoveryThread: Discovers clients on the network by sending out IGMP
           general queries.
7      PubSubThread: Listens for redis expiry messages and removes expired devices.
8
9  """
10 import time
11 import threading
12 import logging
13 logging.getLogger("scapy.runtime").setLevel(logging.ERROR)
14 # suppresses following message
15 # WARNING: No route found for IPv6 destination :: (no default route?)
16 from scapy.all import conf, sendp, ARP, Ether, ETHER_BROADCAST, IP
17 from scapy.contrib.igmp import IGMP
18 import util
19
20
21 class ARPDiscoveryThread(threading.Thread):
22     """This thread is used to discover clients on the network by sending ARP
           requests."""
23
24     def __init__(self, gateway, network):
25         """Initialises the thread.
```

```
26
27          Args:
28              gateway (str): The gateways IP address.
29              network (str): The network IP address.
30
31          """
32          threading.Thread.__init__(self)
33          self.gateway = gateway
34          self.network = network
35
36      def run(self):
37          """Sends broadcast ARP requests for every possible client of the network.
38          Received ARP replies are processed by a SniffThread.
39          """
40          sendp(Ether(dst=ETHER_BROADCAST) / ARP(op=1, psrc=self.gateway, pdst=self.
                network))
41
42
43  class IGMPDiscoveryThread(threading.Thread):
44      """This thread is used to discover clients on the network by sending IGMP
            general queries."""
45
46      _IGMP_MULTICAST = "224.0.0.1"
47      """str: Multicast address used to send IGMP general queries."""
48      _SLEEP = 60
49      """int: Time to wait before sending packets anew."""
50      _IGMP_GENERAL_QUERY = 0x11
51      """int: Value of type Field for IGMP general queries."""
52      _TTL = 1
53      """int: Value for TTL for IP packet."""
54
55      def __init__(self, gateway, network, ip, mac):
56          """Initialises the thread.
57
58          Args:
59              gateway (str): The gateway's IP address.
60              network (str): The network IP address.
61              mac (str): MAC address of this device.
62              ip (str): IP address of this device.
63
64          """
```

```
65          threading.Thread.__init__(self)
66          self.gateway = gateway
67          self.network = network
68          self.mac = mac
69          self.ip = ip
70
71      def run(self):
72          """Sends IGMP general query packets using the multicast address 224.0.0.1.
73          Received replies are processed by a SniffThread.
74          """
75
76          # create IGMP general query packet
77          ether_part = Ether(src=self.mac)
78          ip_part = IP(ttl=self._TTL, src=self.ip, dst=self._IGMP_MULTICAST)
79          igmp_part = IGMP(type=self._IGMP_GENERAL_QUERY)
80
81          # Called to explicitely fixup associated IP and Ethernet headers
82          igmp_part.igmpize(ether=ether_part, ip=ip_part)
83
84          while True:
85              sendp(ether_part / ip_part / igmp_part)
86              time.sleep(self._SLEEP)
87
88
89  class PubSubThread(threading.Thread):
90      """This thread is used to listen for redis expiry keyspace event messages."""
91
92      __SUBSCRIBE_TO = "__keyevent@{}__:expired"
93      """Used to subscribe to the keyspace event expired."""
94
95      def __init__(self, redis, logger):
96          """Initialises the thread.
97
98          Args:
99              redis (apate_redis.ApateRedis): Used for obtaining the required PubSub
                    object.
100             logger (logging.Logger): Used to log messages.
101
102         """
103         threading.Thread.__init__(self)
104         self.redis = redis
```

```
105         self.logger = logger
106         self.pubsub = self.redis.get_pubsub()
107 
108     def run(self):
109         """Subscribes to redis expiry keyspace events and removes the ip address of
                the expired device from the network set."""
110         self.pubsub.subscribe(self.__SUBSCRIBE_TO.format(self.redis.__DB))
111         for message in self.pubsub.listen():
112             self.logger.debug("Removed expired device {} from network {}".format(
                    util.get_device_ip(message['data']), util.get_device_net(message['
                    data'])))
113             # removes the ip of the expired device (the removed device entry) from
                    the network set
114             self.redis._del_device_from_network(util.get_device_ip(message['data']),
                     util.get_device_net(message['data']))
115 
116     def stop(self):
117         """Closes the connection of the PubSub object."""
118         self.pubsub.close()
```

A.10.6. lib/sniff_thread.py

```
1  # coding=utf-8
2  """This module provides several classes that are used to implement
3  a thread that listens to incoming ARP packets.
4 
5  Classes:
6      _SniffThread: Abstract class, that should be inherited.
7      HolisticSniffThread: Inherits _SniffThread and implements the holistist spoofing
            listener.
8      SelectiveSniffThread: Inherits _SniffThread and implements the selective
           spoofing listener.
9 
10 """
11 import thread
12 import logging
13 import threading
14 
15 logging.getLogger("scapy.runtime").setLevel(logging.ERROR)
16 # suppresses following message
17 # WARNING: No route found for IPv6 destination :: (no default route?)
```

```
18 from scapy.all import sendp, ARP, Ether, IP, ETHER_BROADCAST, sniff
19 from scapy.contrib.igmp import IGMP
20
21 import util
22
23
24 class _SniffThread(threading.Thread):
25     """This is an abstract class, which should be inherited to define the
26     behaviour fo the sniffing thread."""
27
28     _DELAY = 7.0
29     """float: Delay after which packets are sent."""
30     _SNIFF_FILTER = "arp and inbound"
31     """str: tcpdump filter used for scapy's sniff function."""
32     _LFILTER = staticmethod(lambda x: x.haslayer(ARP))
33     """function: lambda filter used for scapy's sniff function."""
34
35     def __init__(self, interface, gateway, mac, gate_mac):
36         """Initialises several things needed to define the thread's behaviour.
37
38         Args:
39             interface (str): The network interface which should be used. (e.g. eth0)
40             gateway (str): IP address of the gateway.
41             mac (str): MAC address of the spoofing device. (own MAC address)
42             gate_mac (str): MAC address of the gateway.
43
44         """
45         threading.Thread.__init__(self)
46         self.interface = interface
47         self.gateway = gateway
48         self.mac = mac
49         self.gate_mac = gate_mac
50
51     def run(self):
52         """Starts sniffing for incoming ARP packets with scapy.
53         Actions after receiving a packet ar defines via _packet_handler.
54         """
55         # the filter argument in scapy's sniff function seems to be applied too late
56         # therefore some unwanted packets are processed (e.g. tcp packets of ssh
               session)
```

```
57          # but it still decreases the number of packets that need to be processed by
                the lfilter function
58          sniff(prn=self._packet_handler, filter=self._SNIFF_FILTER, lfilter=self.
                _LFILTER, store=0, iface=self.interface)
59
60      def _packet_handler(self, pkt):
61          """This method should be overriden to define the thread's behaviour."""
62          pass
63
64      @staticmethod
65      def stop():
66          """May be used to kill the thread, if it is not a daemon thread."""
67          thread.exit()
68
69
70  class HolisticSniffThread(_SniffThread):
71      """Implements the abstract class _SniffThread and also implements
72      the listener of the holistic spoofing mode of Apate.
73      """
74
75      def __init__(self, interface, gateway, mac, gate_mac):
76          """Initialises several things needed to define the thread's behaviour.
77
78          Args:
79              interface (str): The network interface which should be used. (e.g. eth0)
80              gateway (str): IP address of the gateway.
81              mac (str): MAC address of the spoofing device. (own MAC address)
82              gate_mac (str): MAC address of the gateway.
83
84          """
85          super(self.__class__, self).__init__(interface, gateway, mac, gate_mac)
86
87      def _packet_handler(self, pkt):
88          """This method is called for each packet received through scapy's sniff
                function.
89          Incoming ARP requests are used to spoof involved devices.
90
91          Args:
92              pkt (str): Received packet via scapy's sniff (through socket.recv).
93          """
94          # when ARP request
```

```
95          if pkt[ARP].op == 1:
96
97              # packets intended for this machine (upribox)
98              if pkt[Ether].dst == self.mac:
99                  # incoming packets(that are sniffed): Windows correctly fills in the
                        hwdst, linux (router) only 00:00:00:00:00:00
100                 # this answers packets asking if we are the gateway (directly not
                        via broadcast)
101                 # Windows does this 3 times before sending a broadcast request
102                 sendp(Ether(dst=pkt[Ether].src) / ARP(op=2, psrc=pkt[ARP].pdst, pdst
                        =pkt[ARP].psrc, hwdst=pkt[ARP].hwsrc, hwsrc=self.mac))
103
104             # broadcast request to or from gateway
105             elif pkt[Ether].dst.lower() == util.hex2str_mac(ETHER_BROADCAST) and (
                    pkt[ARP].psrc == self.gateway or pkt[ARP].pdst == self.gateway):
106                 # spoof transmitter
107                 packets = [Ether(dst=pkt[Ether].src) / ARP(op=2, psrc=pkt[ARP].pdst,
                        pdst=pkt[ARP].psrc, hwsrc=self.mac, hwdst=pkt[ARP].hwsrc)]
108
109                 # get mac address of original target
110                 dest = self.gate_mac
111                 if pkt[ARP].pdst != self.gateway:
112                     # send arp request if destination was not the gateway
113                     dest = util.get_mac(pkt[ARP].pdst, self.interface)
114
115                 if dest:
116                     # spoof receiver
117                     packets.append(Ether(dst=dest) / ARP(op=2, psrc=pkt[ARP].psrc,
                            hwsrc=self.mac, pdst=pkt[ARP].pdst, hwdst=dest))
118
119                 # some os didn't accept an answer immediately (after sending the
                        first ARP request after boot
120                 # so, send packets after some delay
121                 threading.Timer(self._DELAY, sendp, [packets]).start()
122
123
124 class SelectiveSniffThread(_SniffThread):
125     """Implements the abstract class _SniffThread and also implements
126     the listener of the selective spoofing mode of Apate.
127     """
128
```

```
129     _SNIFF_FILTER = "(arp or igmp) and inbound"
130     """str: tcpdump filter used for scapy's sniff function."""
131     _LFILTER = staticmethod(lambda x: any([x.haslayer(layer) for layer in (ARP, IGMP
            )]))
132     """function: lambda filter used for scapy's sniff function."""
133
134     def __init__(self, interface, gateway, mac, gate_mac, redis):
135         """Initialises several things needed to define the thread's behaviour.
136
137         Args:
138             interface (str): The network interface which should be used. (e.g. eth0)
139             gateway (str): IP address of the gateway.
140             mac (str): MAC address of the spoofing device. (own MAC address)
141             gate_mac (str): MAC address of the gateway.
142             redis (apate_redis.ApateRedis): Used to add new devices to redis db.
143
144         """
145         super(self.__class__, self).__init__(interface, gateway, mac, gate_mac)
146         self.redis = redis
147
148     def _packet_handler(self, pkt):
149         """This method is called for each packet received through scapy's sniff
                function.
150
151         Args:
152             pkt (str): Received packet via scapy's sniff (through socket.recv).
153         """
154
155         if pkt.haslayer(ARP):
156             self._arp_handler(pkt)
157         elif pkt.haslayer(IGMP):
158             self._igmp_handler(pkt)
159
160     def _arp_handler(self, pkt):
161         """"This method is called for each incoming ARP packet received through
                scapy's sniff function.
162         Incoming ARP requests are used to spoof involved devices and add new devices
163         to the redis db. New devices are also added if ARP replies are received.
164
165         Args:
166             pkt (str): Received packet via scapy's sniff (through socket.recv).
```

```
167         """
168         # when ARP request
169         if pkt[ARP].op == 1:
170             # packets intended for this machine (upribox)
171             if pkt[Ether].dst == self.mac:
172                 # incoming packets(that are sniffed): Windows correctly fills in the
                        hwdst, linux (router) only 00:00:00:00:00:00
173                 # this answers packets asking if we are the gateway (directly not
                        via broadcast)
174                 # Windows does this 3 times before sending a broadcast request
175                 sendp(Ether(dst=pkt[Ether].src) / ARP(op=2, psrc=pkt[ARP].pdst, pdst
                        =pkt[ARP].psrc, hwdst=pkt[ARP].hwsrc, hwsrc=self.mac))
176                 # add transmitting device to redis db
177                 self.redis.add_device(pkt[ARP].psrc, pkt[ARP].hwsrc)
178
179             # broadcast request to or from gateway
180             elif pkt[Ether].dst.lower() == util.hex2str_mac(ETHER_BROADCAST) and (
                    pkt[ARP].psrc == self.gateway or pkt[ARP].pdst == self.gateway):
181                 # spoof transmitter
182                 packets = [Ether(dst=pkt[Ether].src) / ARP(op=2, psrc=pkt[ARP].pdst,
                        pdst=pkt[ARP].psrc, hwsrc=self.mac, hwdst=pkt[ARP].hwsrc)]
183
184                 # get mac address of original target
185                 dest = self.gate_mac
186                 if pkt[ARP].pdst != self.gateway:
187                     # send arp request if destination was not the gateway
188                     dest = util.get_mac(pkt[ARP].pdst, self.interface)
189
190                 if dest:
191                     # spoof receiver
192                     packets.append(Ether(dst=dest) / ARP(op=2, psrc=pkt[ARP].psrc,
                            hwsrc=self.mac, pdst=pkt[ARP].pdst, hwdst=dest))
193
194                 # add transmitting device to redis db
195                 self.redis.add_device(pkt[ARP].psrc, pkt[ARP].hwsrc)
196                 # add receiving device to redis db
197                 self.redis.add_device(pkt[ARP].pdst, dest)
198
199                 # some os didn't accept an answer immediately (after sending the
                        first ARP request after boot
200                 # so, send packets after some delay
```

```
201                 threading.Timer(self._DELAY, sendp, [packets]).start()
202         else:
203             # ARP reply
204             # add transmitting device to redis db
205             self.redis.add_device(pkt[ARP].psrc, pkt[ARP].hwsrc)
206
207     def _igmp_handler(self, pkt):
208         """This method is called for each IGMP packet received through scapy's
                sniff function.
209         Incoming IGMP answers are used to spoof involved devices and add new devices
210         to the redis db.
211
212         Args:
213             pkt (str): Received packet via scapy's sniff (through socket.recv).
214         """
215         # if util.get_mac(pkt[IP].src,self.interface):
216         self.redis.add_device(pkt[IP].src, pkt[Ether].src)
217         sendp([Ether(dst=pkt[Ether].src) / ARP(op=2, psrc=self.gateway, pdst=pkt[IP
                ].src, hwdst=pkt[Ether].src),
218                Ether(dst=self.gate_mac) / ARP(op=2, psrc=pkt[IP].src, pdst=self.
                    gateway, hwdst=self.gate_mac)])
```

A.10.7. lib/util.py

```
1  # coding=utf-8
2  """Provides several useful functions used by other modules."""
3  import logging
4
5  logging.getLogger("scapy.runtime").setLevel(logging.ERROR)
6  # suppresses following message
7  # WARNING: No route found for IPv6 destination :: (no default route?)
8  from scapy.all import srp, ARP, Ether, ETHER_BROADCAST
9
10
11 def hex2str_mac(hex_val):
12     r"""Converts a hex mac address into a human-readable string representation.
13
14     Example:
15         "\x11\x22\x33\x44\x55\x66"  -->  "11:22:33:44:55:66"
16
17     Args:
```

```
18          hex_val (str): String containg a mac address as hex values.
19
20      Results:
21          str: Human-readably MAC address.
22
23      """
24      return ':'.join(["{:02x}".format(ord(x)) for x in hex_val])
25
26
27  def get_mac(ip, interface):
28      """Returns the according MAC address for the provided IP address.
29
30      Args:
31          ip (str): IP address used to get MAC address.
32          interface (str): Interface used to send ARP reuest.
33
34      Results:
35          According MAC address as string (11:22:33:44:55:66)
36          or None if no answer has been received.
37      """
38      ans, unans = srp(Ether(dst=ETHER_BROADCAST) / ARP(pdst=ip), timeout=2, iface=
            interface, inter=0.1, verbose=0)
39      for snd, rcv in ans:
40          return rcv.sprintf(r"%Ether.src%")
41
42
43  def get_device_enabled(redis_device):
44      """Returns the enabled part of a device entry."""
45      return redis_device.rsplit(":", 1)[-1]
46
47
48  def get_device_ip(redis_device):
49      """Returns the ip address part of a device entry."""
50      return redis_device.rsplit(":", 2)[-2]
51
52
53  def get_device_net(redis_device):
54      """Returns the network address part of a device entry."""
55      return redis_device.split(":", 3)[2]
```

A.10.8. requirements.txt

```
argparse==1.2.1
distribute==0.6.24
docutils==0.12
hiredis==0.2.0
lockfile==0.12.2
netaddr==0.7.18
netifaces==0.10.4
python-daemon==2.1.1
redis==2.10.5
scapy==2.3.2
wsgiref==0.1.2
```

A.11. upribox_interface

A.11.1. urls.py

```
"""webapp URL Configuration

The `urlpatterns` list routes URLs to views. For more information please see:
    https://docs.djangoproject.com/en/1.8/topics/http/urls/
Examples:
Function views
    1. Add an import:  from my_app import views
    2. Add a URL to urlpatterns:  url(r'^$', views.home, name='home')
Class-based views
    1. Add an import:  from other_app.views import Home
    2. Add a URL to urlpatterns:  url(r'^$', Home.as_view(), name='home')
Including another URLconf
    1. Add an import:  from blog import urls as blog_urls
    2. Add a URL to urlpatterns:  url(r'^blog/', include(blog_urls))
"""
from django.conf.urls import include, url
from django.contrib.auth import views as auth_views
from django.views.generic.base import RedirectView

urlpatterns = [

    # www config
```

```
23      url(r'^help/$', "www.views.faq", name="upri_faq"),
24
25      # more config
26      url(r'^more/$', "more.views.more_config", name="upri_more"),
27      url(r'^more/ssh/toggle$', "more.views.ssh_toggle", name="upri_ssh_toggle"),
28      url(r'^more/apate/toggle$', "more.views.apate_toggle", name="upri_apate_toggle")
            ,
29
30      # Auth config
31      url(r'^login/$', auth_views.login, {"template_name": "login.html"}, name="
            upri_login"),
32      url(r'^logout/$', auth_views.logout, {"next_page": "upri_login"}, name="
            upri_logout"),
33
34      # WLAN config
35      url(r'^$', RedirectView.as_view(pattern_name='upri_silent', permanent=False),
            name='upri_index'),
36      url(r'^silent/$', "wlan.views.silent", name="upri_silent"),
37      url(r'^ninja/$', "wlan.views.ninja", name="upri_ninja"),
38      url(r'^ninja/toggle/$', "wlan.views.ninja_toggle", name="upri_ninja_toggle"),
39      url(r'^ninja/check_pi3/$', "wlan.views.check_pi3", name="upri_check_pi3"),
40      url(r'^jobstatus/$', "www.views.jobstatus", name="upri_jobstatus"),
41      url(r'^jobstatus/clear/$', "www.views.clear_jobstatus", name="
            upri_clear_jobstatus"),
42
43      # VPN config
44      url(r'^vpn/$', "vpn.views.vpn_config", name="upri_vpn"),
45      url(r'^vpn/toggle/$', "vpn.views.vpn_toggle", name="upri_vpn_toggle"),
46      url(r'^vpn/check_connection/$', "vpn.views.check_connection", name="
            upri_check_connection"),
47      url(r'^vpn/generate/$', "vpn.views.vpn_generate", name="upri_vpn_generate"),
48      url(r'^vpn/delete/(?P<slug>\w+)$', "vpn.views.vpn_delete", name="upri_vpn_delete
            "),
49      url(r'^vpn/(?P<download_slug>\w+)/upribox.ovpn$', "vpn.views.vpn_download", name
            ="upri_vpn_download"),
50      url(r'^vpn/createlink/(?P<slug>\w+)$', "vpn.views.vpn_create_download", name="
            upri_vpn_create_download"),
51      url(r'^vpn/get/(?P<slug>\w+)$', "vpn.views.vpn_get", name="upri_vpn_get"),
52
53      # statistics config
```

```
54      url(r'^statistics/$', "statistics.views.get_statistics", name="upri_statistics")
            ,
55      url(r'^statistics/get$', "statistics.views.json_statistics", name="
            upri_get_statistics"),
56
57  ]
```

A.11.2. more/views.py

```
1   # -*- coding: utf-8 -*-
2   from __future__ import unicode_literals
3   from django.template import RequestContext
4   from django.shortcuts import render_to_response
5   from django.contrib.auth.decorators import login_required
6   from django.http import Http404, HttpResponse
7   from lib import jobs
8   from lib.info import UpdateStatus
9   from .forms import AdminForm
10  import logging
11  from django.contrib.auth.models import User
12  from . import jobs as sshjobs
13  from django.core.urlresolvers import reverse
14
15  # Get an instance of a logger
16  logger = logging.getLogger('uprilogger')
17
18  @login_required
19  def more_config(request):
20      context = RequestContext(request)
21
22      if request.method == 'POST':
23
24          form = AdminForm(request, request.POST)
25
26          if form.is_valid():
27              new_password = form.cleaned_data['password2']
28              new_username = form.cleaned_data['username']
29
30              old_password = form.cleaned_data['oldpassword']
31              old_username = request.user.username
32
```

```
33              logger.info("updating user %s..." % old_username)
34              u = User.objects.get(username=old_username)
35
36              #sanity check, this should never happen
37              if not u:
38                  logger.error("unexpected error: user %s does not exist" %
                        old_username)
39                  return HttpResponse(status=500)
40
41              u.set_password(new_password)
42              u.username = new_username
43              u.save()
44              logger.info("user %s updated to %s (password changed: %s)" % (
                    old_username, new_username, new_password != old_password) )
45              context.push({'message': True})
46
47          else:
48              logger.error("admin form validation failed")
49
50      else:
51          form = AdminForm(request)
52
53      update_status = UpdateStatus()
54
55      context.push({
56          'form': form,
57          'messagestore': jobs.get_messages(),
58          'update_time': update_status.update_utc_time,
59          'version': update_status.get_version()
60      })
61
62      return render_to_response("more.html", context)
63
64  @login_required
65  def ssh_toggle(request):
66      if request.method != 'POST':
67          raise Http404()
68
69      state = request.POST['enabled']
70      jobs.queue_job(sshjobs.toggle_ssh, (state,))
71
```

```
72      return render_to_response("modal.html", {"message": True, "refresh_url": reverse
            ('upri_more')})
73
74  @login_required
75  def apate_toggle(request):
76      if request.method != 'POST':
77          raise Http404()
78
79      state = request.POST['enabled']
80      jobs.queue_job(sshjobs.toggle_apate, (state,))
81
82      return render_to_response("modal.html", {"message": True, "refresh_url": reverse
            ('upri_more')})
```

A.11.3. more/jobs.py

```
1   # -*- coding: utf-8 -*-
2   from __future__ import unicode_literals
3   import lib.jobs as jobs
4   import lib.utils as utils
5   from django.utils.translation import ugettext as _
6   import logging
7   logger = logging.getLogger('uprilogger')
8
9
10  def toggle_ssh(state):
11
12      if state in ['yes', 'no']:
13          try:
14              if state == 'yes':
15                  jobs.job_message(_("SSH wird gestartet..."))
16              else:
17                  jobs.job_message(_("SSH wird gestoppt..."))
18
19              logger.debug("restarting ssh")
20              utils.exec_upri_config('enable_ssh', state)
21              utils.exec_upri_config('restart_ssh')
22              jobs.job_message(_("Konfiguration von SSH erfolgreich."))
23
24          except utils.AnsibleError as e:
25              logger.error("ansible failed with error %d: %s" % (e.rc, e.message))
```

```
26              if state == 'yes':
27                  jobs.job_message(_("Starten von SSH fehlgeschlagen."))
28              else:
29                  jobs.job_message(_("Stoppen von SSH fehlgeschlagen."))
30      else:
31          jobs.job_message(_("Es ist ein unbekannter Fehler aufgetreten."))
32
33  def toggle_apate(state):
34
35      if state in ['yes', 'no']:
36          try:
37              if state == 'yes':
38                  jobs.job_message(_("Apate ARP Spoofing Daemon wird gestartet..."))
39              else:
40                  jobs.job_message(_("Apate ARP Spoofing Daemon wird gestoppt..."))
41
42              logger.debug("restarting apate")
43              utils.exec_upri_config('enable_apate', state)
44              utils.exec_upri_config('restart_apate')
45              jobs.job_message(_("Konfiguration von Apate ARP Spoofing Daemon
                    erfolgreich."))
46
47          except utils.AnsibleError as e:
48              logger.error("ansible failed with error %d: %s" % (e.rc, e.message))
49              if state == 'yes':
50                  jobs.job_message(_("Starten von Apate ARP Spoofing Daemon
                        fehlgeschlagen."))
51              else:
52                  jobs.job_message(_("Stoppen von Apate ARP Spoofing Daemon
                        fehlgeschlagen."))
53      else:
54          jobs.job_message(_("Es ist ein unbekannter Fehler aufgetreten."))
```

A.11.4. more/templates/more.html

```
1  {% extends request.is_ajax|yesno:"base_ajax.html,base.html" %}
2
3  {% load i18n %}
4  {% load widget_tweaks %}
5  {% load base_extras %}
6
```

```
7  {% block title %}{% trans "Admin - upribox" %}{% endblock %}
8
9  {% block header %}
10     <h1>{% trans "Admin" %}</h1>
11     <p>{% trans "Admin-Daten &auml;ndern." %}</p>
12 {% endblock %}
13
14 {% block content %}
15
16     {% trans "Admin-Zugangsdaten" as form_title %}
17     {% url "upri_more" as href %}
18     {% include "form.html" %}
19
20     <h2>{% trans "SSH" %}</h2>
21     <p>{% trans "Mithilfe dieser Funktion können erfahrene Benutzer die upribox
           selbstständig konfigurieren." %}</p>
22     <form>
23         <fieldset>
24             <legend>{% trans "Ein-/Ausschalten" %}</legend>
25
26             {% get_fact 'ssh' 'general' 'enabled' as sshenabled%}
27
28             {% if sshenabled == 'yes'%}
29                 <div class="switch icon i-on">
30                     <p>{% trans "SSH ist aktiviert" %}</p>
31                     <button class="js-toggle-button" data-state-enabled="no" href="
                           {% url 'upri_ssh_toggle' %}">{% trans "Ausschalten" %}</
                           button>
32                 </div>
33             {% else %}
34                 <div class="switch icon i-off">
35                     <p>{% trans "SSH ist deaktiviert" %}</p>
36                     <button class="js-toggle-button" data-state-enabled="yes" href="
                           {% url 'upri_ssh_toggle' %}">{% trans "Einschalten" %}</
                           button>
37                 </div>
38             {% endif %}
39         </fieldset>
40     </form>
41
42     <h2>{% trans "Apate" %}</h2>
```

```
43      <p>{% trans "Mithilfe dieser Funktion können erfahrene Benutzer den Apate ARP
            Spoofing Daemon konfigurieren." %}</p>
44      <form>
45          <fieldset>
46              <legend>{% trans "Ein-/Ausschalten" %}</legend>
47
48              {% get_fact 'apate' 'general' 'enabled' as apateenabled%}
49
50              {% if apateenabled == 'yes'%}
51                  <div class="switch icon i-on">
52                      <p>{% trans "Apate ist aktiviert" %}</p>
53                      <button class="js-toggle-button" data-state-enabled="no" href="
                            {% url 'upri_apate_toggle' %}">{% trans "Ausschalten" %}</
                            button>
54                  </div>
55              {% else %}
56                  <div class="switch icon i-off">
57                      <p>{% trans "Apate ist deaktiviert" %}</p>
58                      <button class="js-toggle-button" data-state-enabled="yes" href="
                            {% url 'upri_apate_toggle' %}">{% trans "Einschalten" %}</
                            button>
59                  </div>
60              {% endif %}
61          </fieldset>
62      </form>
63
64       <h2>{% trans "upribox Software" %}</h2>
65      <p>{% trans "Letztes update: " %} {{ update_time|date:"SHORT_DATETIME_FORMAT" }}
             (UTC)</p>
66      <p>Version: {{ version }}</p>
67
68  {% endblock %}
```

A.12. django/files/upri-config.py

```
1  #!/usr/bin/env python
2  import json
3  import sys
4  import subprocess
5  from jsonmerge import merge
```

```
6  from os import path
7  import sys
8  sys.path.insert(0, "/usr/share/nginx/www-upri-interface/lib/")
9  import passwd
10 import ssid
11 import domain
12 import re
13 from datetime import datetime
14 from urlparse import urlparse
15 import os
16 import sqlite3
17
18 # directory where facts are located
19 FACTS_DIR = "/etc/ansible/facts.d"
20 # path to the ansible-playbook executeable
21 ANSIBLE_COMMAND = "/usr/local/bin/ansible-playbook"
22 # path to the used inventory
23 ANSIBLE_INVENTORY = "/var/lib/ansible/local/environments/production/inventory_pull"
24 # path to the used playbook
25 ANSIBLE_PLAY = "/var/lib/ansible/local/local.yml"
26 # path to the openvpn client config template
27 CLIENT_TEMPLATE = "/etc/openvpn/client_template"
28
29 #
30 # revokes previously generated openvpn client certificates
31 # return values:
32 # 26 failed to revoke certificate
33 def action_delete_profile(slug):
34     try:
35         filename = os.path.basename(slug)
36
37         rc = subprocess.call(['/usr/bin/openssl', 'ca', '-revoke', '/etc/openvpn/ca
               /%sCert.pem' % filename])
38         rc = subprocess.call(['/usr/bin/openssl', 'ca', '-gencrl', '-crlhours', '1',
                '-out', '/etc/openssl/demoCA/crl.pem'])
39
40         #os.remove('/etc/openvpn/ca/%sKey.pem' % filename)
41         #os.remove('/etc/openvpn/ca/%sCert.pem' % filename)
42
43     except Exception as e:
44         print "failed to delete client files"
```

```
45          print str(e)
46          return 26
47
48      return 0
49
50  #
51  # generate openvpn client certificates and saves the
52  # generated openvpn client config into the database
53  # return values:
54  # 16: database error
55  # 21: entry does not exists in database
56  # 24: provided domain is not valid
57  # 23: unable to create client certificate files
58  # 22: openvpn client template is missing
59  def action_generate_profile(profile_id):
60      with open('/etc/ansible/default_settings.json', 'r') as f:
61          config = json.load(f)
62
63      dbfile = config['django']['db']
64
65      try:
66          conn = sqlite3.connect(dbfile)
67          c = conn.cursor()
68          c.execute("SELECT slug,dyndomain FROM vpn_vpnprofile WHERE id=?",(profile_id
                ,))
69          data = c.fetchone()
70          if not data:
71              #invalid profile id
72              print 'profile id does not exist in database'
73              return 21
74
75          slug = data[0]
76          dyndomain = data[1]
77
78          if not check_domain(dyndomain):
79              return 24
80
81          dyndomain = domain.Domain(data[1]).get_match()
82
83          filename = os.path.basename(slug)
84
```

```
85          rc = subprocess.call(['/usr/bin/openssl', 'req', '-newkey', 'rsa:2048', '-
                nodes', '-subj', "/C=AT/ST=Austria/L=Vienna/O=Usable Privacy Box/OU=VPN/
                CN=%s" % filename, '-keyout', '/etc/openvpn/ca/%sKey.pem' % filename, '-
                out', '/etc/openvpn/ca/%sReq.pem' % filename])
86
87          if rc != 0:
88              print "error while creating client certificate reques"
89              return 23
90
91          subprocess.call(['/usr/bin/openssl', 'ca', '-in', '/etc/openvpn/ca/%sReq.pem
                ' % filename, '-days', '730', '-batch', '-out', '/etc/openvpn/ca/%sCert.
                pem' % filename, '-notext', '-cert', '/etc/openvpn/ca/caCert.pem', '-
                keyfile', '/etc/openvpn/ca/caKey.pem'])
92
93          if rc != 0:
94              print "error while creating client certificate"
95              return 23
96
97          os.remove('/etc/openvpn/ca/%sReq.pem' % filename)
98
99          if os.path.isfile(CLIENT_TEMPLATE):
100             with open(CLIENT_TEMPLATE, 'r') as template, open('/etc/openvpn/ca/%sKey
                    .pem' % filename, 'r') as client_key, open('/etc/openvpn/ca/%sCert.
                    pem' % filename, 'r') as client_cert:
101                 temp = template.read()
102                 temp = temp.replace("#CLIENT_KEY", client_key.read())
103                 temp = temp.replace("#CLIENT_CERT", client_cert.read())
104                 temp = temp.replace("<IP-ADRESS>", dyndomain)
105
106                 c.execute("UPDATE vpn_vpnprofile SET config=? where id=?", (temp,
                        profile_id))
107                 conn.commit()
108         else:
109             print "client template is missing"
110             return 22
111
112         conn.close()
113     except Exception as e:
114         print "failed to write to database"
115         print str(e)
116         return 16
```

```
117     return 0
118
119 #
120 # parse the privoxy logfiles and insert data into django db
121 # return values:
122 # 16: database error
123 # 20: new entries have been added
124 def action_parse_logs(arg):
125     rlog = re.compile('(\d{4}-\d{2}-\d{2} (\d{2}:?){3}).\d{3} [a-z0-9]{8} Crunch:
            Blocked: (.*)')
126
127     with open('/etc/ansible/default_settings.json', 'r') as f:
128         config = json.load(f)
129
130     dbfile = config['django']['db']
131     logfile = os.path.join(config['log']['general']['path'], config['log']['privoxy'
            ]['subdir'], config['log']['privoxy']['logfiles']['logname'])
132
133     if os.path.isfile(logfile):
134         print "parsing privoxy logfile %s" % logfile
135         with open(logfile, 'r') as privoxy:
136             logentries = []
137             for line in privoxy:
138                 try:
139                     res = re.match(rlog, line)
140                     if res:
141                         sdate = res.group(1)
142                         ssite = res.group(3)
143                         pdate = datetime.strptime(sdate, '%Y-%m-%d %H:%M:%S')
144                         psite = urlparse(ssite).netloc
145                         logentries.append( (psite,pdate) )
146                         print "found new block: [%s] %s" % (sdate,psite)
147                 except Exception as e:
148                     print "failed to parse line \"%s\": %s" % (line, e.message)
149
150         #write updates into db
151         if len(logentries) > 0:
152             try:
153                 conn = sqlite3.connect(dbfile)
154                 c = conn.cursor()
```

```
155                 c.executemany("INSERT INTO statistics_privoxylogentry(url,log_date)
                        VALUES (?,?)", logentries)
156                 c.execute("DELETE FROM statistics_privoxylogentry WHERE log_date <=
                        date('now','-6 month')")
157                 conn.commit()
158                 conn.close()
159                 # delete logfile
160                 os.remove(logfile)
161                 # todo: implement reload
162                 subprocess.call(["/usr/sbin/service", "privoxy", "restart"])
163             except Exception as e:
164                 print "failed to write to database"
165                 return 16
166             return 20
167
168     else:
169         print "failed to parse privoxy logfile %s: file not found" % logfile
170         return 0
171
172 #
173 # set a new ssid for the upribox "silent" wlan
174 # return values:
175 # 12: ssid does not meet policy
176 #
177 def action_set_ssid(arg):
178     print 'setting ssid to "%s"' % arg
179     if not check_ssid(arg):
180         return 12
181     ssid = { "upri": { "ssid": arg } }
182     write_role('wlan', ssid)
183
184 # return values:
185 # 11: password does not meet password policy
186 def action_set_password(arg):
187     print 'setting password'
188     if not check_passwd(arg):
189         return 11
190     passwd = { "upri": { "passwd": arg } }
191     write_role('wlan', passwd)
192
193 #
```

```
194 # return values:
195 # 12: ssid does not meet policy
196 #
197 def action_set_tor_ssid(arg):
198     print 'setting tor ssid to "%s"' % arg
199     if not check_ssid(arg):
200         return 12
201     ssid = { "ninja": { "ssid": arg } }
202     write_role('wlan', ssid)
203
204 # return values:
205 # 11: password does not meet password policy
206 def action_set_tor_password(arg):
207     print 'setting tor password'
208     if not check_passwd(arg):
209         return 11
210     passwd = { "ninja": { "passwd": arg } }
211     write_role('wlan', passwd)
212
213 def action_restart_wlan(arg):
214     print 'restarting wlan...'
215     return call_ansible('ssid')
216
217 # return values:
218 # 10: invalid argument
219 def action_set_tor(arg):
220     if arg not in ['yes', 'no']:
221         print 'error: only "yes" and "no" are allowed'
222         return 10
223     print 'tor enabled: %s' % arg
224     passwd = { "general": { "enabled": arg } }
225     write_role('tor', passwd)
226
227 def action_restart_tor(arg):
228     print 'restarting tor...'
229     return call_ansible('toggle_tor')
230
231 # return values:
232 # 10: invalid argument
233 def action_set_vpn(arg):
234     if arg not in ['yes', 'no']:
```

```
235         print 'error: only "yes" and "no" are allowed'
236         return 10
237     print 'vpn enabled: %s' % arg
238     vpn = { "general": { "enabled": arg } }
239     write_role('vpn', vpn)
240     return 0
241
242 def action_restart_vpn(arg):
243     print 'restarting vpn...'
244     #return 0 # TODO implement
245     return call_ansible('toggle_vpn')
246
247 # return values:
248 # 10: invalid argument
249 def action_set_ssh(arg):
250     if arg not in ['yes', 'no']:
251         print 'error: only "yes" and "no" are allowed'
252         return 10
253     print 'ssh enabled: %s' % arg
254     en = { "general": { "enabled": arg } }
255     write_role('ssh', en)
256
257 def action_restart_ssh(arg):
258     print 'restarting ssh...'
259     return call_ansible('toggle_ssh')
260
261 # return values:
262 # 10: invalid argument
263 def action_set_apate(arg):
264     if arg not in ['yes', 'no']:
265         print 'error: only "yes" and "no" are allowed'
266         return 10
267     print 'apate enabled: %s' % arg
268     en = { "general": { "enabled": arg } }
269     write_role('apate', en)
270
271 def action_restart_apate(arg):
272     print 'restarting apate...'
273     return call_ansible('toggle_apate')
274
275 def check_passwd(arg):
```

```
276     pw = passwd.Password(arg)
277     if not pw.is_valid():
278         if not pw.has_digit():
279             print 'the password must contain at least 1 digit'
280         if not pw.has_lowercase_char():
281             print 'the password must contain at least 1 lowercase character'
282         if not pw.has_uppercase_char():
283             print 'the password must contain at least 1 uppercase character'
284         if not pw.has_symbol():
285             print 'the password must contain at least 1 special character'
286         if not pw.has_allowed_length():
287             print 'the password must be between 8 to 63 characters long'
288         if not pw.has_only_allowed_chars():
289             print 'the password must only contain following special characters: %s'
                % pw.get_allowed_chars()
290
291         return False
292     else:
293         return True
294
295 def check_ssid(arg):
296     ssid_value = ssid.SSID(arg)
297     if not ssid_value.is_valid():
298         if not ssid_value.has_allowed_length():
299              print 'the password must be between 1 to 32 characters long'
300         if not ssid_value.has_only_allowed_chars():
301             print 'the ssid must only contain following special characters: %s' %
                ssid_value.get_allowed_chars()
302
303         return False
304     else:
305         return True
306
307 def check_domain(arg):
308     domain_value = domain.Domain(arg)
309     if not domain_value.is_valid():
310         if not domain_value.has_allowed_length():
311              print 'the password can only contain up to 255 characters'
312         if not domain_value.has_only_allowed_chars():
313             print 'the domain must only contain following special characters: %s' %
                domain_value.get_allowed_chars()
```

```
314
315         return False
316     else:
317         return True
318
319 def action_restart_network(arg):
320     print 'restarting network...'
321     #return 0 # TODO implement
322     return call_ansible('network_config')
323
324 # add your custom actions here
325 ALLOWED_ACTIONS = {
326     'set_ssid': action_set_ssid,
327     'set_password': action_set_password,
328     'set_tor_ssid': action_set_tor_ssid,
329     'set_tor_password': action_set_tor_password,
330     'restart_wlan': action_restart_wlan,
331     'enable_tor': action_set_tor,
332     'restart_tor': action_restart_tor,
333     'enable_vpn': action_set_vpn,
334     'restart_vpn': action_restart_vpn,
335     'enable_ssh': action_set_ssh,
336     'restart_ssh': action_restart_ssh,
337     'enable_apate': action_set_apate,
338     'restart_apate': action_restart_apate,
339     'parse_logs': action_parse_logs,
340     'generate_profile': action_generate_profile,
341     'delete_profile': action_delete_profile,
342     'restart_network': action_restart_network
343 }
344
345 #
346 # calls ansible and executes the given tag locally
347 #
348 def call_ansible(tag):
349     return subprocess.call([ANSIBLE_COMMAND, '-i', ANSIBLE_INVENTORY, ANSIBLE_PLAY,
        "--tags", tag, "--connection=local"])
350
351 #
352 # write the custom json "data" to the fact with the given name "rolename"
353 #
```

```
354 def write_role(rolename, data):
355     p = path.join(FACTS_DIR, rolename + '.fact')
356     try:
357         with open(p, 'r') as data_file:
358             js = json.load(data_file)
359     except IOError:
360         js = {}
361 
362     js = merge(js, data)
363     with open(p, 'w+') as data_file:
364         json.dump(js, data_file, indent=4)
365 
366 # return values:
367 # 0: ok
368 # 1: syntax error
369 # 2: invalid number of arguments
370 # 3: invalid action
371 def main():
372     # append empty second parameter if none given
373     if len(sys.argv) == 2:
374         sys.argv.append('')
375 
376     if len(sys.argv) !=3:
377         usage(2)
378 
379     action = sys.argv[1]
380     args = sys.argv[2]
381 
382     # check if requested action is valid
383     if sys.argv[1] in ALLOWED_ACTIONS:
384         print "action: %s" % action
385         return ALLOWED_ACTIONS[action](args)
386     else:
387         usage(3)
388 
389 def usage(ex):
390     print "usage: %s <action> <args>" % sys.argv[0]
391     print "allowed actions:"
392     for action in ALLOWED_ACTIONS:
393         print "    %s" % action
394     exit(ex)
```

```
395
396
397 if __name__ == "__main__":
398     exit(main())
```

Bibliography

[1] J. R. Mayer and J. C. Mitchell, "Third-party web tracking: Policy and technology," in *2012 IEEE Symposium on Security and Privacy*. IEEE, 2012, pp. 413–427.

[2] P. Wood, B. Nahorney, K. Chandrasekar, S. Wallace, K. Haley *et al.*, "Symantec Internet Security Threat Report," *Volume*, vol. 21, 2016.

[3] CENTRE FOR INTERNATIONAL GOVERNANCE INNOVATION & IPSOS. (2014) Cigi-ipsos global survey on internet security and trust. (last access: 08.02.2016). [Online]. Available: https://www.cigionline.org/internet-survey

[4] E. W. Felten and J. A. Halderman, "Digital rights management, spyware, and security," *IEEE Computer Society*, 2006.

[5] X. d. C. de Carnavalet and M. Mannan, "Killed by proxy: Analyzing client-end tls interception software," in *Network and Distributed System Security Symposium (NDSS 2016), San Diego, CA, USA*, 2016.

[6] K. Kopper, *The Linux Enterprise Cluster: build a highly available cluster with commodity hardware and free software*. No Starch Press, 2005.

[7] M. Schwartzkopff, *Clusterbau: Hochverfügbarkeit mit pacemaker, OpenAIS, heartbeat und LVS: hochverfügbare Linux-Server*. O'Reilly Germany, 2010.

[8] The OpenBSD Project. OpenBSD PF: Firewall Redundancy (CARP and pfsync). (last access: 04.07.2016). [Online]. Available: https://www.openbsd.org/faq/pf/carp.html

[9] D. C. Plummer, "Ethernet address resolution protocol: Or converting network protocol addresses to 48.bit ethernet address for transmission on ethernet hardware," Internet Requests for Comments, RFC Editor, STD 37, November 1982, http://www.rfc-editor.org/rfc/rfc826.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc826.txt

[10] K. R. Fall and W. R. Stevens, *TCP/IP illustrated, volume 1: The protocols*. addison-Wesley, 2011, ch. 4, pp. 165–180.

[11] R. Philip, "Securing wireless networks from arp cache poisoning," Master's thesis, 2007.

[12] N. R. Samineni, F. A. Barbhuiya, and S. Nandi, "Stealth and semi-stealth mitm attacks, detection and defense in ipv4 networks," in *Parallel Distributed and Grid Computing (PDGC), 2012 2nd IEEE International Conference on*. IEEE, 2012, pp. 364–367.

[13] A. Lockhart, *Network security hacks*. " O'Reilly Media, Inc.", 2006.

[14] D. Bruschi, A. Ornaghi, and E. Rosti, "S-arp: a secure address resolution protocol," in *Computer Security Applications Conference, 2003. Proceedings. 19th Annual*. IEEE, 2003, pp. 66–74.

[15] W. Lootah, W. Enck, and P. McDaniel, "Tarp: Ticket-based address resolution protocol," *Computer Networks*, vol. 51, no. 15, pp. 4322–4337, 2007.

[16] S. Y. Nam, D. Kim, J. Kim *et al.*, "Enhanced arp: preventing arp poisoning-based man-in-the-middle attacks," *IEEE communications letters*, vol. 14, no. 2, pp. 187–189, 2010.

[17] C. Neuman, T. Yu, S. Hartman, and K. Raeburn, "The kerberos network authentication service (v5)," Internet Requests for Comments, RFC Editor, RFC 4120, July 2005, http://www.rfc-editor.org/rfc/rfc4120.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc4120.txt

[18] B. Bakhache and R. Rostom, "Kerberos secured address resolution protocol (karp)," in *Digital Information and Communication Technology and its Applications (DICTAP), 2015 Fifth International Conference on*. IEEE, 2015, pp. 210–215.

[19] St. Pölten University of Applied Sciences. Institute of IT Security Research. (last access: 04.07.2016). [Online]. Available: https://www.fhstp.ac.at/en/research/institute-of-it-security-research

[20] ——. Usable Privacy Box (upribox). (last access: 04.07.2016). [Online]. Available: https://www.fhstp.ac.at/de/forschung/projekte/usable-privacy-box-upribox

[21] Markus Huber. usableprivacy/upribox. (last access: 04.07.2016). [Online]. Available: https://github.com/usableprivacy/upribox

[22] R. T. Fielding, J. Gettys, J. C. Mogul, H. F. Nielsen, L. Masinter, P. J. Leach, and T. Berners-Lee, "Hypertext transfer protocol – http/1.1," Internet Requests for Comments,

RFC Editor, RFC 2616, June 1999, http://www.rfc-editor.org/rfc/rfc2616.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc2616.txt

[23] R. Dingledine, N. Mathewson, and P. Syverson, "Tor: The second-generation onion router," DTIC Document, Tech. Rep., 2004.

[24] Red Hat, Inc. Ansible Documentation. (last access: 04.07.2016). [Online]. Available: https://docs.ansible.com/ansible/index.html

[25] Django Software Foundation. Django: The Web framework for perfectionists with deadlines. (last access: 04.07.2016). [Online]. Available: https://www.djangoproject.com/

[26] Philippe Biondi. Scapy. (last access: 19.07.2016). [Online]. Available: http://www.secdev.org/projects/scapy/

[27] Biondi, Philippe and Raynal, Fren and Martini, Sebastien and Kacherginsky, Peter and Loss, Dirk. Scapy v2.1.1-dev documentation. (last access: 19.07.2016). [Online]. Available: http://www.secdev.org/projects/scapy/doc/

[28] Sanfilippo, Salvatore. Introduction to Redis. (last access: 19.07.2016). [Online]. Available: http://redis.io/topics/introduction

[29] ——. Redis Keyspace Notifications. (last access: 19.07.2016). [Online]. Available: http://redis.io/topics/notifications

[30] T. Narten, E. Nordmark, W. Simpson, and H. Soliman, "Neighbor discovery for ip version 6 (ipv6)," Internet Requests for Comments, RFC Editor, RFC 4861, September 2007, http://www.rfc-editor.org/rfc/rfc4861.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc4861.txt

[31] R. M. Hinden and S. E. Deering, "Ip version 6 addressing architecture," Internet Requests for Comments, RFC Editor, RFC 2373, July 1998, http://www.rfc-editor.org/rfc/rfc2373.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc2373.txt

[32] B. Cain, S. Deering, I. Kouvelas, B. Fenner, and A. Thyagarajan, "Internet group management protocol, version 3," Internet Requests for Comments, RFC Editor, RFC 3376, October 2002, http://www.rfc-editor.org/rfc/rfc3376.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc3376.txt

[33] G. Wissowa, *Paulys Realencyclopädie der classischen Altertumswissenschaft*. Metzler-Verlag, 1894, vol. I, 2.

[34] B. Aboba, D. Thaler, and L. Esibov, "Link-local multicast name resolution (llmnr)," Internet Requests for Comments, RFC Editor, RFC 4795, January 2007.

[35] S. Cheshire and M. Krochmal, "Multicast dns," Internet Requests for Comments, RFC Editor, RFC 6762, February 2013, http://www.rfc-editor.org/rfc/rfc6762.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc6762.txt

[36] M. Anagnostopoulos, G. Kambourakis, P. Kopanos, G. Louloudakis, and S. Gritzalis, "Dns amplification attack revisited," *Computers & Security*, vol. 39, pp. 475–485, 2013.

[37] Kelley, Simon. DNSMASQ. (last access: 04.07.2016). [Online]. Available: http://www.thekelleys.org.uk/dnsmasq/docs/dnsmasq-man.html

[38] Y. Rekhter, R. G. Moskowitz, D. Karrenberg, G. J. de Groot, and E. Lear, "Address allocation for private internets," Internet Requests for Comments, RFC Editor, BCP 5, February 1996, http://www.rfc-editor.org/rfc/rfc1918.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc1918.txt

[39] ISO, "Linux Standard Base (LSB) core specification 3.1 – Part 1: Generic specification," International Organization for Standardization, Geneva, CH, Standard, 2006.

[40] S. McCanne and V. Jacobson, "The bsd packet filter: A new architecture for user-level packet capture." in *USENIX winter*, vol. 46, 1993.

[41] Dirk Loss and Sébastien Mainand and Pierre Lalet and Guillaume Valadona and Alex Chan. Download and Installation. (last access: 04.07.2016). [Online]. Available: https://github.com/secdev/scapy/blob/88c1dbac32008bd2af772bb088f93106231b9f94/doc/scapy/installation.rst