Open vSwitch Configuration for Separation of KVM/libvirt VMs

Jonas Andre, Johannes Naab*
*Chair of Network Architectures and Services, Department of Informatics
Technical University of Munich, Germany
Email: andre@in.tum.de, naab@net.in.tum.de

Abstract—**Virtual machines are very useful tools to provide configured computers for a lecture. These machines are usually connected via a layer two network to the Internet. As the virtual machines cannot be trusted because they can send arbitrary frames, the network and other virtual machines need to be protected of network attacks. This paper presents a solution and implementation to improve the security of libvirt virtual machines connected via an Open vSwitch. The outcome is an Open Flow implementation which prevents virtual machines from attacks like spoofing, Denial of Service, and Man in the Middle attacks.**

Index Terms—**Open vSwitch, libvirt, Open Flow, virtual machine security, layer 2 attacks**

1. Introduction

Lectures like *Grundlagen Rechnernetze und Verteilte Systeme* and *Advanced Computer Networks* at TUM provide virtual machines (VMs) to their students. On those machines they perform practical experiments. Using VMs is comfortable as it provides each student the same configuration where the experiments work out of the box.
For the experiments, the machines need to be connected to the Internet. This is done via a single layer two network connecting all VMs with their gateway. As all machines are within a single layer two network, there are attacks which could not only harm the VM of a student, but all VMs within the network.
This work deals with security problems of multiple VMs within one layer two network. The paper specializes on VMs virtualized by the Linux KVM module, configured by using the libvirt API, where the network is managed via the Open vSwitch module.
As discovered in previously running environments with 900 VMs, the CPU workload should be addressed. ARP requests need to be processed at all machines parallel. With this number of VMs and a high number of ARP requests, the CPU load of the host grew significantly. A solution to prevent ARP broadcasting is also presented.
The paper is structured as follows. In Section 2, popular (layer two) attacks which are relevant for the used scenario are considered. Requirements to prohibit those attacks are determined. Afterwards, the section describes the CPU workload problem related to the network. In Section 3, the current implementation is described and evaluated. The new implementation for the libvirt/KVM scenario with Open vSwitch is described in Section 4. The conlusion in Section 5 evaluates the implemented rules and shows investigations for future work.

2. Security Issues in Layer 2 Networks

In this section, relevant security issues within layer two networks are explained. A short description of possible attacks is given. It also shows the theoretical possibilities to prevent those attacks. Authenticity is not given for sender and receiver addresses within layer 2 network frames. Consequently, there are several spoofing attacks on which a VM can send with fake IP and MAC addresses.

MAC Spoofing. If a VM wants to hide what packets it is sending, it can use MAC spoofing. In this attack the sender machine does not send an Ethernet frame with the source MAC of its own interface. Instead, a fake MAC address or the address of another machine within the network is used. With this it is not possible anymore to track the sender of the frame. To be able to determine the true sender of a frame we must ensure that the VMs can only send frames with their own (predefined) MAC address. To prevent MAC Spoofing the following requirement needs to be fulfilled.

Req.1 VMs can only send Ethernet frames from their configured MAC address

IP Spoofing. Another spoofing attack is IP/IPv6 spoofing. Here, the attacker uses a fake IP address to hide its identity to hosts outside of this layer two network. The problem is that attacks on hosts outside of the layer two network identify the attacker by its IP address. This address can be tracked back to the IP network of the VMs. If the address was spoofed, it is not possible to find the attacker within the network. This should be possible, as the responsible VM for such an attack needs to be found. To achieve this two requirements need to be met:

Req.2 VMs can only send IPv4 frames from their configured IPv4 address
Req.3 VMs can only send IPv6 messages from their configured IPv6 addresses

ARP Spoofing. A well known attack which may result in a DoS or a MitM is the so called ARP spoofing [1]. ARP is designed to find the MAC address of a machine within a layer two network by its IPv4 address [2]. On an ARP request resolving a specific IP address the machine with the configured IP address should answer with a reply containing the MAC address of its interface on which the IP address is configured. However, any machine could answer on an ARP request. If a machine answers to an ARP request which is not meant for it, future IP packets will be routed to the spoofing machine instead to its correct destination. The following requirement protects the network from APR spoofing:

Req.4 VMs can only answer with ARP requests containing their configured IP address

IPv6 Router Spoofing. A Router Solicitation is a message that is sent from a host to find routers within its layer two network [3]. Listening routers answer with router advertisements which include information about the network configuration. VMs can manipulate the interface configuration of other VMs. This is possible by sending Router Advertisements by themselves. As this can lead to DoS and MitM attacks [4], the messages must be forbidden.

Req.5 VMs are not allowed to send Router Advertisements

Neighbor Discovery Spoofing. In IPv6 NDP also takes the place of ARP in IPv4. Spoofing is here possible in the same way as it is described before. To prevent spoofing, the machines are only allowed to answer to Neighbor Solicitations meant for them. This also addresses a problem with Stateless Address Auto Configuration (SLAAC) [5]. With SLAAC IPv6 interfaces generate their own IP address. In our setup SLAAC is used to configure link local IP addresses. One step is to check whether the IP address generated is already used. For this, a Duplicate Address Detection (DAD) message is sent. This is a Neighbor Solicitation message for the generated address. If no one answers to this message, the address can be assigned to the interface. A typical attack in DAD is a machine answering to these DAD messages although it is not configured with this address. This leads to a DoS as the VM cannot generate an IPv6 address.

Req.6 VMs can only send Neighbor Advertisements containing one of their configured IPv6 address

ICMP Redirect. Another possible attack is the abuse of the Redirect message [4]. This message is usually used to inform clients that a router which received the packet knows a better route to the destination IP. By sending such messages, the VMs can be manipulated to send packets to other VMs instead of the gateway.

Req.7 VMs are not allowed to send the Redirect messages

Broadcast Flooding. As ARP requests are broadcasted in Ethernet networks [2], every VM gets all ARP request. Every machine needs to process the request to decide whether it needs to response to the request or not. Hundreds of VMs processing ARP requests at the same time leads to high CPU consumption. The reason for the high CPU consumption lies in the Spectre and Meltdown security fixes. As the VMs have to be loaded and unloaded every time to prevent reading uncleaned memory of the other machines. This needs to be done for every incoming ARP request. The workload for loading and unloading the machines is high. There should be a solution such that the ARP requests do not need to be sent to all machines. This is possible because all IP addresses of the virtual machines are known in advance.

Req.8 Prevent ARP broadcasting by sending ARP requests directly to the correct machine

DHCP. The IP addresses of the VMs are known in advance. Nevertheless, a DHCP server is useful to assign IP addresses to the machines. An attacker can fake a DHCP server on a virtual machine by answering packets destined for the original DHCP server or sending DHCP offers to other clients [6]. This should be denied as this may lead to DoS or MitM for other VMs. DHCP server messages can be dropped by filtering on the DHCP server UDP source port.

Req.9 VMs are not allowed to send DHCP server messages

3. Current State

In this section, the setup of the network connecting the VMs is described. It also shows how the machines are created and configured as this is the basis on which the countermeasures against network attacks can be piggybacked. At the end of this section the defenses installed currently are evaluated.

There is one Open vSwitch which connects all VMs with the gateway to the Internet. The Open vSwitch with name `vm-switch` is shown in Figure 1. Traffic from and to the

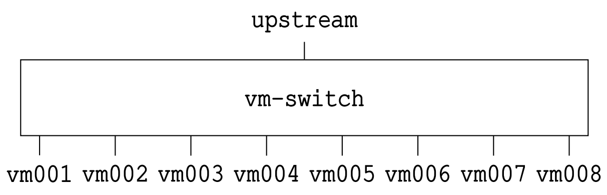

Figure 1: The switch and its ports

Internet goes through interface `upstream`. All VMs are connected via an interface `vmXXX` where `XXX` is the number of the VM.

The VMs are created via the command `virt install`. A script called `create-student.sh` generates one or multiple VMs for a user. To be able to access the generated information, name and address(es) of the interface(s) are encoded within the *metadata* option of the `virt install` command.

The necessary security rules are created via Open Flow. A script for generating these rules is triggered by a QEMU hook. QEMU is the machine emulator on which KVM is based on. On every start of a VM the – within the `virt install` defined – *metadata* is parsed. With this information the script finds the port identifier of the Open vSwitch on which the VM is connected to `vm-switch`. The MAC and IPv4 address of the interface are also extracted from the *metadata* file. With information about port ID, MAC address, and IPv4 address, Open Flow rules are installed on the switch. It is intended to create rules to prevent MAC and ARP spoofing. The following rules are created with the *ovs-ofctl create-flow* command within the QEMU hook. To provide better readability, the `in_port=$port` which is part of every rule is omitted.

```
dl_src=$mac priority=40 action=normal
dl_src=* priority=39 action=drop
arp arp_sha=$mac nw_src=$ip4 priority=40
    ↪action=normal
arp arp_sha=$mac nw_src=* priority=39 action=
    ↪drop
```

Listing 1: Installed rules

The rules shown in Listing 1 are written in Open Flow syntax. They are intended to work the following way:

- Allow frames from the given port where the MAC address matches the configured MAC address of the machine
- Drop all frames with another MAC address
- If the packet is an ARP response, allow only responses of the configured IP and MAC address
- Drop all other ARP responses

During the implementation of security features (explained in Section 4) within this work, it was detected that these rules do not work in the intended way. The problem lies in the first rule. As it has the same priority as rule number three, it is not deterministic which of those rules is considered at first [7]. Therefore, if an ARP response with the configured source MAC address and a spoofed IP address is sent the action depends on the chosen rule. If the third rule is used, everything is fine. In case the first rule is used the switch determines a correct sender MAC address and performs the action `normal`. This means the packet is processed like it is a normal unconfigured switch, i.e., it is sent to the given destination. With this ARP spoofing is possible. An attacker using this ARP spoofing attack can act as a Man in the Middle for packets which are destined for the spoofed IP address. This is the case as all packets are first sent to the attacker because the sender expects the IP address at the attacker.
At the moment the following issues are not considered

- Defenses against attacks based on IPv6
- DHCP attacks
- DoS by spamming broadcasts
- DoS by ARP request which may lead to CPU overconsumption as all machines have to process the packets
- Cleanup of old rules when a VM is destroyed

4. Implementation

This section is about how the issues addressed in Section 2 are implemented. The changes in the script for creating the students VMs and how the security relevant rules are installed are pointed out.
The script for creating the VMs was updated to handle different new functionalities. Configured IPv6 addresses of the machines are now also written to the *metadata* of the machines. This is necessary for the automatic creation of security rules.
These rules are created via a hook which is triggered by the QEMU creation of the VMs. It runs during step *start* and phase *begin*. The basic functionality of the script is to create new security rules for each VM and delete those rules when the machine is deleted or shut down. To implement the security rule in the Open vSwitch, the `ovs-ofctl add flows` command is used.
When the script is started, the data written to the *metadata* file is parsed. As only the information about the different interfaces MAC, IPv4, and IPv6 global unique addresses are given, the link local addresses of the interfaces need to be derived. With the specified interface name in the *metadata* of the VM, the Open vSwitch port ID of the machines gateway interface can be found. This port ID is necessary to generate input-dependent flow rules on the switch. For convenience and an easy way to delete the created rules, rules are tagged with an unique identifier (called `cookie`) of the machines. This is the MAC address of the interface connecting the VM to the switch.
Open vSwitch manages its OpenFlow rules with different tables. Each table consists of several rules with different priority. A table needs to have an entry with priority `0`. This rule is the default rule (table miss rule). In an Open vSwitch with default configuration there exists only one table called **table0**. This has only the default rule with action `normal` which indicates the switch should handle the frame like a typical switch. Beside `normal` there exist more actions like `drop` (dropping the frame), `goto_table:X`, where `X` is the id of the table in which the frame should be tested next, and `X` which sends the frame to port ID `X` [7].
The implemented rule set is defined in four tables. An overview of the structure can be seen in Figure 2. It shows which table handles the specific requirements.

`table0`	`table1`	`table2`	`table3`

`MAC spoof`
`ARP`
	`IP spoof`
	`ARP spoof`
	`DHCP spoof`
		`Direct ARP`	`ICMP spoof`

Figure 2: The Open Flow tables of vm-switch

The following source code shows the installed rules of the different tables. The `cookie` is not represented below for providing better readability. If no `in_port` is specified within a rule, it means this rule is only applied on the incoming port where the VM is connected to. The entry table **table0** prevents MAC spoofing and initiates the ARP optimization.

```
dl_src=$mac priority=40 action=goto_table:1
priority=39 action=drop
in_port=* arp priority=1 action=goto_table:2
in_port=* priority=0 action=normal
```

Listing 2: Installed rules in `table0`

The first rule of Listing 2 defines that frames coming from the connected machine which are sent with the correct MAC address are processed further in **table1**. The next rule (it has lower priority) drops all packets coming to the switch which are not sent from the defined MAC address. It matches against all MAC addresses. However, the correct address is already tested in the previous rule. With these two rules, Req. 1 as specified in Section 2 is met. All packets that do not enter the switch from a virtual machine (interface `upstream`) are first matched against the third rule. This rule does not specify an `in_port`. It tests if the frame is an ARP packet. Then it is further processed in the ARP optimization table (**table2**). All other packets from `upstream` match against the table miss entry which defines to process the frame with action `normal`.

```
arp arp_sha=$mac arp_spa=$ip4 priority=40
    ↪action=table2
```

```
2   udp udp_src=67 priority=39 action=drop
3   ip nw_src=$ip4 priority=38 action=normal
4   icmp6 ipv6_src=$ip6 priority=37 action=table3
5   icmp6 ipv6_src=$eui64 priority=36 action=
        ↪table3
6   icmp6 ipv6_src=:: priority=35 action=table3
7   udp6 udp_src=547 priority=34 action=drop
8   ipv6 ipv6_src=$ip6 priority=33 action=normal
9   ipv6 ipv6_src=$eui64 priority=32 action=
        ↪normal
10  in_port=* priority=0 action=drop
```

Listing 3: Installed rules `table1`

In the second table, ARP spoofing, DHCP server spoofing, and IP(v6) spoofing are considered. The rule with highest priority (in Line 1 of Listing 3) only allows ARP requests and responses with non-spoofed addresses (Req. 4). These frames are also (like the ARP rule in **table0**) sent to **table2** which does ARP optimization. Rules number two and seven prevent DHCP server messages from VMs by dropping all packets that are sent from the DHCP server UDP port [8], [9]. This prevents the machines from faking a DHCP server (Req. 9). All other IPv4 packets are allowed by action `normal` in the next rule if the source address matches the configured one (Req. 2). The next three flows handle ICMPv6 spoofing. It is only allowed to send ICMP messages from one of its own IPv6 addresses (Req. 6). Additionally, the unspecified address needs to be allowed to permit Duplicate Address Detection. All these ICMPv6 messages are sent to **table3** to handle more ICMP security issues. Rules eight and nine allow all other IPv6 packets that leave the virtual machine with the correct link local or global IPv6 address (Req. 3).

```
1   in_port=* arp arp_op=1 arp_tpa=$ip4
        ↪priority=40 action=$port
2   in_port=1 arp arp_op=1 priority=2 action=drop
3   in_port=* arp arp_op=1 priority=1 action=1
4   in_port=* priority=0 action=normal
```

Listing 4: Installed rules `table2`

Now the rules of the ARP optimization table (**table2**) are explained. The first rule within Listing 4 defines that all ARP requests that are destined for any of the virtual machines are not broadcasted like ARP is usually done, but they are only sent directly to the corresponding machine (here no `in_port` is considered). Such a rule is created for every VM because of the VM's IP address within it. ARP request which come from the interface `upstream`, i.e., the Internet and were not matched against the first rule are dropped as these are destined to IP addresses which are not present within this network. All other ARP requests coming from one of the VMs are sent to `upstream`. All other ARP packets are processed the normal way (by defining the table miss with action `normal`). With this table Req. 8 is met.

```
1   icmp_type=134 priority=40 action=drop
2   icmp_type=136 nd_target=$ip6 priority=39
        ↪action=normal
3   icmp_type=136 nd_target=$eui64 priority=38
        ↪action=normal
4   icmp_type=136 priority=37 action=drop
5   icmp_type=137 priority=36 action=drop
6   in_port=* priority=0 action=normal
```

Listing 5: Installed rules `table3`

The fourth table represented in Listing 5 handles spoofing within ICMPv6. First, all router advertisements (ICMP type 134) sent from the VMs are dropped (Req. 5). This prevents the machines from faking to be a router to others in the network. The next three rules only allow sending Neighbor Advertisement messages from their own IPv6 addresses. All other Neighbor Advertisements are dropped (Req. 6). Additionally, all ICMPv6 Redirect messages are forbidden as this may lead to DoS attacks. With this rule, Req. 7 is also fulfilled.
In the qemu hooks file, the shutdown of the machines is also handled. The deletion of the rules is handled during step *stopped* and phase *end*. Here, all rules affecting the VM are deleted. This is done by matching the cookie of the flows to the MAC address of the VM.

5. Conclusion and Future Work

With the outcome of this paper, the network connecting the VMs is more secure. The implementation handles MAC, IPv4, IPv6, APR, NDP, and DHCP spoofing. Furthermore, the optimization of ARP requests aims for reducing the CPU load of the host system.
As the number of different layer two network attacks grows and new attacks are created over time, more security features need to be added in future. Currently, the problem of tiny IPv6 fragments is not addressed yet [10]. Additionally, the performance of checking the OpenFlow rules can be evaluated in future. An interesting point is whether the checks can be more performant if other strategies (blacklisting, whiteliting) are used. As multicasts and broadcasts can lead to a network overload, a meaningful prevention of this should be explored. A first idea would be rate limiting of the VMs when sending to much traffic into the network. Whether this can be realised is part of future work.

References

[1] S. Whalen, *An Introduction to ARP Spoofing*. Chocobospore, 2001.

[2] D. Plummer, "An Ethernet Address Resolution Protocol or Converting Network Protocol Address to 48.bit Ethernet Address for Transmissing on Ethernet Hardware," RFC 826, 1982.

[3] T. Narten, "Neighbor Discovery for IP version 6 (IPv6)," RFC 4861, 2007.

[4] A. Pilihanto, *A Complete Guide on IPv6 Attack and Defense*. SANS Institute, 2011. Available at https://www.sans.org/reading-room/whitepapers/detection/complete-guide-ipv6-attack-defense-33904.

[5] S. Thomson, "IPv6 Stateless Address Autoconfiguration," RFC 4862, 2007.

[6] Y. Bhaiji, *Understanding, Preventing, and Defending Against Layer 2 Attacks*. Cisco, 2007. Available at https://www.cisco.com/c/dam/global/en_ae/assets/exposaudi2009/assets/docs/layer2-attacks-and-mitigation-t.pdf.

[7] Open Networking Foundation, *OpenFlow Switch Specification*, 2012. Available at https://www.opennetworking.org/wp-content/uploads/2013/04/openflow-spec-v1.3.1.pdf.

[8] R. Droms, "Dynamic Host Configuration Protocol," RFC 2131, 1997.

[9] T. Mrugalski, "Dynamic Host Configuration Protocol for IPv6 (DHCPv6)," RFC 8415, 2018.

[10] A. Atlasis, *Attacking IPv6 Implementation Using Fragmentation*. Center for Strategic Cyberspace + Security Science, 2012. Available at https://media.blackhat.com/bh-eu-12/Atlasis/bh-eu-12-Atlasis-Attacking_IPv6-WP.pdf.