

**SECURING VXLAN-BASED OVERLAY NETWORK  
USING SSH TUNNEL**

by

Saravanan Ramesh

A thesis submitted to the Faculty of the University of Delaware in partial fulfillment of the requirements for the degree of Master of Science in Electrical and Computer Engineering

Summer 2017

© 2017 Saravanan Ramesh  
All Rights Reserved

**SECURING VXLAN-BASED OVERLAY NETWORK  
USING SSH TUNNEL**

by

Saravanan Ramesh

Approved: \_\_\_\_\_  
Stephan K. Bohacek, Ph.D.  
Professor in charge of thesis on behalf of the Advisory Committee

Approved: \_\_\_\_\_  
Kenneth E. Barner, Ph.D.  
Chair of the Department of Electrical and Computer Engineering

Approved: \_\_\_\_\_  
Babatunde A. Ogunnaike, Ph.D.  
Dean of the College of Engineering

Approved: \_\_\_\_\_  
Ann L. Ardis, Ph.D.  
Senior Vice Provost for Graduate and Professional Education

## **ACKNOWLEDGMENTS**

It was an incredible learning experience during my master's thesis study at University of Delaware. I thank my advisor Dr. Stephan Bohacek from the bottom of my heart to have believed in me and to have given me the opportunity to do this project under his guidance. He was the best project mentor one could ask for. He gave me complete freedom to explore, learn and apply different approaches to tackle issues in the project and has always been patient and optimistic with my progress. He guided me in the right direction every time I hit a wall and taught me how to approach a problem. His constant support and encouragement was the primary reason that abled me to finish this project successfully.

A special thanks to Dr. Chase Cotton, for being the coolest and most cheerful professor around. He was the key person who always encouraged and gave me opportunities to keep continuing my learning and grow my skills in the field of security. He guided me to take the right courses since my first semester at UD and was always available to help.

All of this wouldn't have been possible without the support of my brother Kishore, who gave me the courage and motivation to start my thesis and morally supporting me throughout. Along with him, I also would like to thank my mom, dad and sister for trusting in me and cheering me up at every stage. A huge thanks to my friends Kartik, Fatema and Suchi for making my research life a bit easy by always being there to help brainstorm issues along with me and providing me with important feedback, which kept me going in the right direction.

## TABLE OF CONTENTS

LIST OF FIGURES .....	vii
ABSTRACT .....	ix
Chapter	
1 INTRODUCTION .....	1
1.1 VXLAN and Advantages of VXLAN .....	2
2 OPENFLOW .....	5
2.1 OpenFlow Messages.....	6
2.2 OpenFlow Table .....	7
2.3 Flow table Operation .....	8
2.3.1 Flow Matching.....	9
2.3.2 Table-miss: .....	9
2.3.3 Flow removal.....	10
2.4 Open vSwitch .....	11
2.4.1 Open vSwitch Key Components .....	12
2.5 Characteristics of Open vSwitch .....	13
3 VXLAN COMMUNICATION .....	15
3.1 Traditional Client Server communication .....	17
3.2 VXLAN Communication between VTEPs .....	18
4 MININET .....	20
4.1 Setting up Mininet .....	22
5 VXLAN TUNNEL CONFIGURATION ON OVERLAY NETWORK .....	24
5.1 Mininet Configuration at Server1 .....	25

5.1.1	Changing IP Configuration for node h1 .....	26
5.1.2	Changing IP Configuration for node h2 .....	27
5.1.3	VXLAN tunnel set-up .....	27
5.1.4	Addition of Flow table entry .....	30
5.1.4.1	Flow entries at Server1 .....	31
5.2	Mininet Configuration at Server2.....	34
5.2.1	Changing IP Configuration for node h1 .....	35
5.2.2	Changing IP Configuration for node h2 .....	36
5.2.3	VXLAN tunnel set-up .....	37
5.2.4	Addition of Flow table entry .....	39
5.2.4.1	Flow entry at Server2 .....	39
5.3	Testing VXLAN tunnel by sending ICMP .....	42
5.3.1	ICMP from 10.0.0.1 and 10.0.0.2 at Server1 .....	42
5.3.2	ICMP from 10.0.0.3 and 10.0.0.4 at Server2 .....	43
6	STUDY OF PACKET FLOW THROUGH UNENCRYPTED VXLAN TUNNEL .....	44
6.1	Wireshark packet capture .....	44
6.2	Packet flow Walkthrough .....	45
7	VXLAN TUNNEL CONFIGURATION WITH SSH .....	48
7.1	Need for Tunnel Security .....	48
7.2	Network topology of VXLAN tunnel through SSH.....	49
7.3	VXLAN Configuration set-up with SSH .....	50
7.3.1	For SSH tunnel 1 from Server1 to Server2 .....	50
7.3.2	For SSH tunnel 2 from server2 to server1 .....	53
7.3.3	Iptables rule .....	54
8	STUDY OF VXLAN PACKET FLOW THROUGH SSH TUNNEL.....	56
8.1	Wireshark packet capture .....	56
8.2	Packet flow Walkthrough with SSH tunnel: .....	58
9	DISCUSSION.....	61
10	CONCLUSION .....	70

11	FUTURE DIRECTIONS.....	71
	BIBLIOGRAPHY .....	73

## LIST OF FIGURES

Figure 2.1: OpenFlow architecture.....	6
Figure 2.2: OpenFlow flow diagram (Image adapted from [9]).....	8
Figure 3.1: VXLAN frame encapsulation (Image adapted from [3]).....	16
Figure 3.2: UDP communication.....	17
Figure 3.3 VXLAN communication.....	19
Figure 4.1: Mininet Topology (Image adapted from [29]).....	23
Figure 5.1: Overlay network set up using VXLAN .....	24
Figure 5.2: Mininet set-up at server1 .....	25
Figure 5.3: Dump command to view mininet topology components for server1 .....	25
Figure 5.4: VM1 Ethernet configuration at server1 .....	26
Figure 5.5 VM2 Ethernet configuration at server1 .....	27
Figure 5.6: OVS bridge and port details at server1 .....	28
Figure 5.7: Addition of VTEP to OVS bridge at server1 .....	29
Figure 5.8: OpenFlow port details of bridge s1 at server1 .....	30
Figure 5.9: Flow table 0 for server1 .....	31
Figure 5.10: Flow table 1 for server1 .....	32
Figure 5.11: Mininet set-up at server2 .....	34
Figure 5.12: Dump command to view mininet topology components for server2.....	34
Figure 5.13: VM1 Ethernet configuration at server2 .....	35
Figure 5.14: VM2 Ethernet configuration at server2 .....	36

Figure 5.15: OVS bridge and port details at server2 .....	37
Figure 5.16: Addition of VTEP to OVS bridge at server2 .....	38
Figure 5.17: OpenFlow port details of bridge s1 at server2 .....	39
Figure 5.18: Flow table 0 for server2 .....	40
Figure 5.19: Flow table 1 for server2 .....	40
Figure 5.20: ICMP test from server1 .....	42
Figure 5.21: ICMP test from server2.....	43
Figure 6.1: Packet capture at unsecured VXLAN tunnel .....	45
Figure 6.2: Packet flow in unsecured VXLAN tunnel (Image adapted from [3]).....	46
Figure 7.1: Network topology of VXLAN tunnel through SSH .....	50
Figure 7.2: iptables rule at server1 .....	55
Figure 7.3: iptables rule at server2 .....	55
Figure 8.1: VXLAN packet capture for SSH tunnel 1 at server1 .....	56
Figure 8.2: VXLAN packet capture for SSH tunnel 1 at server2.....	57
Figure 8.3: Packet flow in VXLAN tunnel secured by SSH (Image adapted from [3]) .....	58
Figure 9.1: TCP through SSH tunnel .....	69

## **ABSTRACT**

This project focuses on utilizing Virtual Extensible Local Area Network (VXLAN), a tunneling protocol used in cloud overlay networks to address the scalability issues in large production environments, and deploying a security measure to uphold VXLAN data integrity against possible infiltrations or snooping that uses data transparency as a leverage to disrupt the communication and launch attacks on the network. Though there are security implementations for VXLAN tunnel traffic over the IP network on physical VXLAN switches and several firewall rules to restrict network access at the industry, there is no implementation of SSH as a secured means for MAC over IP VXLAN communication between two different servers without the need for external firewall or other traditional security mechanisms. The purpose of this project is to deploy encryption mechanism over the VXLAN traffic on public internet for a secured communication. The first step of the project is manually setting up overlay between Open vSwitch virtual switches using VXLAN on both client and server, and second is configuring SSH tunnel between the hosts and channeling the VXLAN traffic through SSH. The VXLAN traffic over internet is unencrypted and prone to data compromise. Securing VXLAN based overlay network using SSH tunnel encrypts the data, thus protecting its integrity.

## **Chapter 1**

### **INTRODUCTION**

With the whole world moving to the cloud for its personal and business needs, cloud computing has not only grown to be the next best thing in the world of internet but also has become a necessity for most of the users. An increase in demand for virtualization has brought about a tremendous expansion of cloud in terms of computing, network and storage. The overall demand for cloud computing in all its guises is estimated to grow 18% in the year of 2017 and to be more in the following years [1]. With such major demand for cloud virtualization, the physical infrastructure of datacenter is required to support a higher number of virtual machines hosted on the servers. An increase in the virtual machines on the physical server will require an increase in the MAC address tables in the switched Ethernet network as each of the Virtual Machine (VM) would have its own MAC address and an IP address [2]. There can be cases where the VMs in the datacenter be grouped in a VLAN, and with huge industries moving to cloud there might be scenarios where thousands of VM needs to be grouped in a VLAN. There may also be cases where a part of the VLAN group is in one datacenter and the other in a different datacenter. For instance, a company who hosts its servers virtually at a datacenter located at New York might expand its business in other parts of the country and might have to host that location's VM in a nearby datacenter. There may also be cases where a datacenter cannot accommodate all the VMs in that region due to physical infrastructure restraint and must move some of its VMs to a different physical server. The multi-tenant environment of the cloud

helps a cloud service provider to offer elastic services to multiple customers over the same physical infrastructure, isolating them completely from each other's traffic. The traditional VLAN only supports 4096 users in a group thus restricting the expansion of cloud infrastructure [2][5][6]. So, for scenarios like mentioned above, there is a requirement for virtualized environments to have a layer 2 network scale across the datacenter and between datacenters.

To tackle the increase in demand for server virtualization and the requirements pointed above, the datacenters had to find a way to allow more VMs to be a part of a same VLAN group and also a way to communicate between VMs of the same VLAN belonging to different datacenters via overlay network. This lead to the requirement of tunneling mechanisms which provide an encapsulation scheme to carry the MAC traffic from each source VM to the destination VM.

### **1.1 VXLAN and Advantages of VXLAN**

VXLAN stands for Virtual Extensible Local Area Network which is one of the proposed encapsulation protocols which helps tunneling of layer 2 over layer 3 infrastructure. This will help increase the scalability of cloud computing environment while logically separating cloud applications with tenants.

VXLAN can help migrate virtual machines over long distance and play an important role in Software-Defined-Networking (SDN) [7].

One of the major characteristics of VXLAN is that it uses larger naming space as compared to regular VLAN. A traditional 802.1q VLAN uses 12 bit space which would only allow  $2^{12}=4096$  users in a segment. This would not be enough for a large cloud computing environment as discussed in the previous section. Whereas VXLAN uses 24 bit space that allows for over  $2^{24}=16,777,216$  VXLAN identifiers, addressing

the problem, where more than a million users can be a part of a same network within the cloud [3][4].

VXLAN allows layer 2 multipath. Traditional layer 2 network uses Spanning Tree Protocol (STP) which doesn't allow active-active forwarding to prevent loops in the network. STP blocks the use of the links to avoid duplicate paths in the network. A big disadvantage of this attribute from a datacenter fabric point of view is that these ports are very expensive to have them sitting idle unless there is a failure in the network. Use of STP thus limits the number of VLANs that could be used, creating a problem in this growing age of virtualization. So in the place of using STP we use layer 3 routing in order to do Equal Cost Multipath (ECMP) to route the packets in IP, from switch 1 to switch 2 for example, to get to the destination host [2][3].

Another advantage of VXLAN over VLAN is it removes the need to have additional physical layer infrastructure. The forwarding table of the switch need not grow with the increase in the number of the VMs behind the host port. Whereas in traditional VLAN the MAC address table of the Top of Rack (ToR) switch that connects to the servers increases in size with increase in the number of VMs being added to the VLAN network, as all the traffic between the VMs traverse through the switch. A regular ToR could connect 24 to 48 servers. A datacenter would consist of enormous number of server racks, so the ToR switch would be required to maintain the MAC address table for each of the VMs across the server racks which in turn would put heavy load on the table capacity. In cases where the table overflows, the switch may stop learning any further MAC addresses until idle entries age out, leading to significant flooding of unknown destination frames. VXLAN switch does not need to store MAC address of all the VMs in its server as it forwards frames based on the

Virtual Network Interface (VNI) number of the VM. All the VMs belonging to the same segment are associated with a VNI number and the switch on receiving the frame forwards it to the destination as instructed by the flow table [2][3].

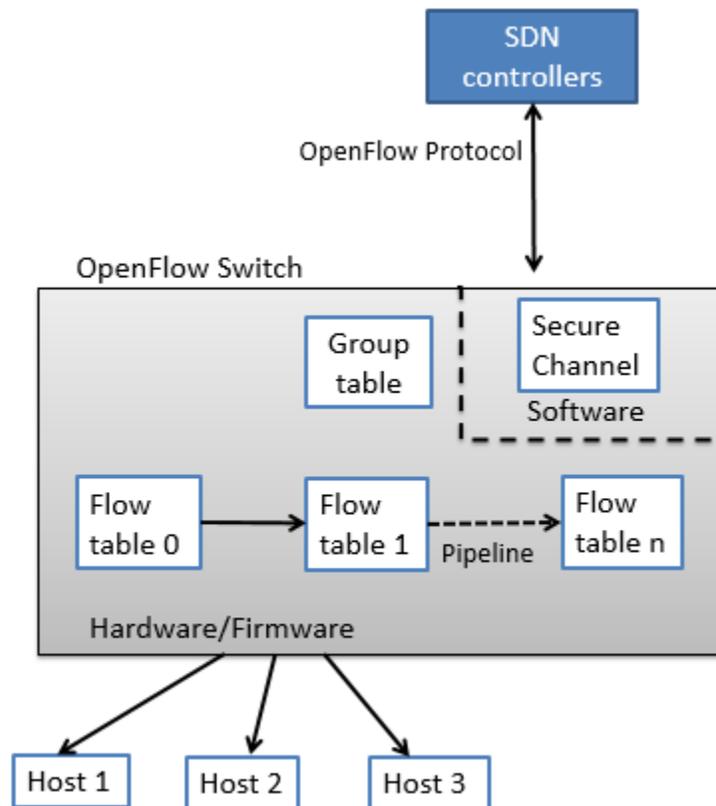
In addition, it also significantly restricts the scope of MAC address duplication of VMs to exist within a VXLAN segment. Two VMs belonging to two different VXLAN segments can have the same MAC addresses as the traffic of each segment is isolated [4].

## **Chapter 2**

### **OPENFLOW**

OpenFlow is the first SDN standard which originally defined the communication interface between control and forwarding layers of the SDN architecture. OpenFlow enabled the SDN controller to communicate directly with both physical and virtual switches and routers in a network aiding it to successfully shape and control the data packets to the required destination from a single point thus achieving holistic enterprise management with a more granular security and a low operational cost [8].

OpenFlow protocol is a standardized protocol for interacting with the forwarding behavior of the switches from multiple vendors. This provides a way to control the behavior of the switches throughout the network dynamically [8].



**Figure 2.1:** OpenFlow architecture

## 2.1 OpenFlow Messages

The OpenFlow protocol supports three message types:

- **Controller-to-switch** messages are initiated by the controller and used to directly manage and inspect the state of the switch.
- **Asynchronous** messages are initiated by the switch and are used to update the controller of the network events and the changes to the switch state.
- **Symmetric** messages are initiated either by the switch or the controller and sent without solicitation.

## 2.2 OpenFlow Table

The OpenFlow table is a data structure that resides in the high-speed data plane of an OpenFlow switch. Its content determines the forwarding behavior and packet handling behavior of that switch. OpenFlow table has one or more flow entries, and each entry has a set of components as listed below [9]:

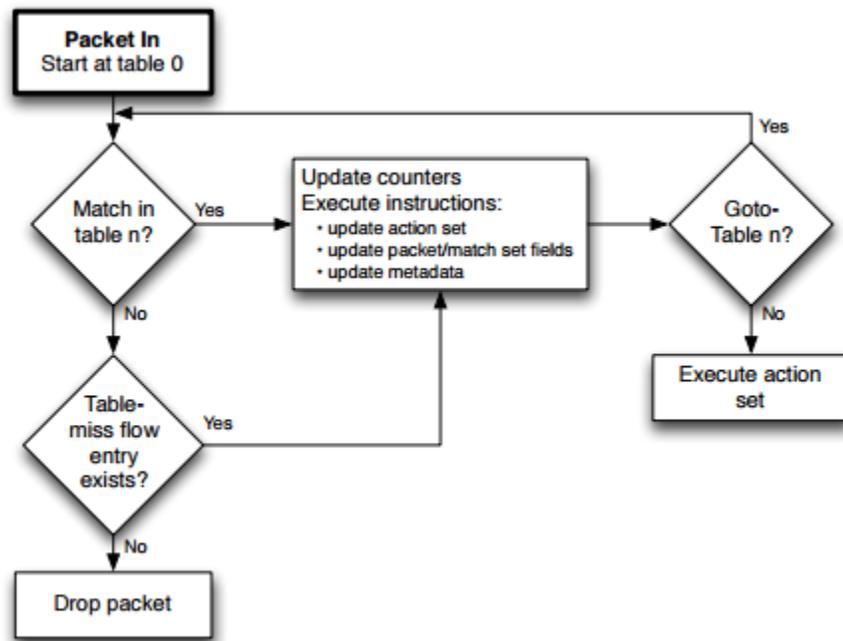
- **Match fields:** are used to identify which packets to perform an action on. They consist of ingress port and packet header optionally metadata specified by the previous port.
- **Priority:** is used to match precedence of the flow entry.
- **Counters:** are updated when packets are matched.
- **Instructions:** are used to modify the action set or pipeline processing.
- **Timeouts:** refers to the maximum amount of time or idle time before flow is expired by the switch.
- **Cookie:** is the opaque data value chosen by the controller. It may be used by the controller to filter flow statistics, flow modification and flow deletion. Not used when processing packets.

A Flow table entry is identified by the matching fields and the priority number. We can have multiple actions per flow table entry. The purpose of the priority column is to resolve conflicts when multiple entries match a particular packet. In scenarios where multiple flow table entries match a particular packet, the rule with the highest priority is the rule that is applied to the packet. A flow entry with a priority of 0 is considered a wildcard entry or a table-miss flow entry.

It is necessary for every OpenFlow switch to have atleast one OpenFlow table. OpenFlow pipeline of every OpenFlow switch consists of multiple flow tables ( as seen in Figure 2.1), each having multiple flow entries. In case of just one flow table the OpenFlow pipeline is simplified. OpenFlow pipeline basically defines how the packets are handled by the switch using flow tables [8][28].

### 2.3 Flow table Operation

A VM belonging to a network having OpenFlow implementation goes through the process as depicted in figure 2.2:



**Figure 2.2:** OpenFlow flow diagram (Image adapted from [9])

### **2.3.1 Flow Matching**

The OpenFlow switch on receiving the packet from the associated VM first performs a table lookup. In case of multiple flow tables, packet flow is determined based on OpenFlow pipelining (refer figure 2.2).

At first the packet match field is extracted from the packet. This match field might vary according to the type of packet. IP address, ingress ports and metadata are some of the possible packet match fields considered.

Only when match field of a packet matches with the flow table entry, the action specified is performed. In scenarios with multiple action sets per match field, each action is performed one after the other. In cases where the packet satisfies multiple table entries, the one with the highest priority is considered over the rest. If a table entry has a match field as ANY, irrespective of the packet type, all possible packet header match fields are considered a match for the flow entry, by default.

The counters of the applied flow entries must be updated for every action performed and the packet is sent to the next  $n^{\text{th}}$  table to repeat the same process until the packet transmits the entire pipeline [9].

### **2.3.2 Table-miss:**

Each flow table must have a default table-miss entry to deal with packets that misses to match any of the flow entries. This table-miss flow entry decides the fate of such packets. This includes forwarding it to the next flow table, sending it over to the controller or dropping it permanently.

A flow entry is considered a table-miss if the packet match field and priority wildcards all the match fields in the table and is given the lowest priority. This occurs when the entries fall outside the range of the matches provided by the flow table. Such

table-miss entries are transmitted to the Controller by using the CONTROLLER reserved port. The controller decides where to transfer the packet next. In case the table-miss flow entry is not available in the flow table then the packet is dropped by default [9].

### **2.3.3 Flow removal**

The flow entries inside the OpenFlow table are removed by any one of the two possible methods. One is by a controller request to drop the packet as unwanted or via the switch flow expiry mechanism where the entry sits idle more than the timeout period [9].

The controller can keep the table-miss entry for a restricted period and can modify it as and when needed in accordance to the network requirement. The controller may also actively remove flow entries from a table by sending `OFPPC_DELETE` or `OFPPC_DELETE_STRICT` commands.

The removal of flow entries through the switch flow expiry mechanism is independent of the controller rule. This occurs in accordance with the `idle_timeout` and `hard_timeout` associated with these entries. The flow table must note the last entry time of the packet that match the flow entry. If the next matching packet doesn't arrive beyond the `idle_timeout` period of the flow entry, then the flow entry is removed from the table. Whereas in case of `hard_timeout`, irrespective of the table receiving the flow entry matching packet or not, the entry is dropped when the `hard_timeout` expires [9][10].

These are the core steps in a simple packet flow in a OpenFlow environment. This process repeats for every packet entering the OpenFlow pipeline. The complexity

of the flow model may vary depending on the type and dynamicity of the network we implement OpenFlow on.

## 2.4 Open vSwitch

Open vSwitch (OVS) is an open source OpenFlow capable virtual switch that is typically used with hypervisors to interconnect virtual machines within a host and between different hosts across networks.

OVS ties together all the virtual machines within a host residing on a server, which makes it critical component in many SDN deployments. Using OVS for multi-tenant network virtualization is considered a core element of various datacenter SDN deployments.

OVS supports many traditional switch features such as VLAN tagging and 802.1q trunking, Standard Spanning Tree Protocol, LACP, port mirroring (SPAN/RSPAN), Flow Export (netflow, sflow, etc), tunneling (GRE, VXLAN, IPSEC), QoS control. In addition, it is designed to support across multiple servers such as Cisco's Nexus 1000V and Linux-based virtualization technologies such as KVM, VirtualBox, Xen/XenServer [11].

Open vSwitch is the first entry point for all the VMs sending traffic to the network and is the ingress point into overlay networks running on top of physical networks in the datacenter. OVS can also not need any assistance from a kernel module and operate entirely in user space [11].

One important feature to be noted about OVS is that it doesn't have a native [SDN Controller](#) or manager, like the Virtual Supervisor Manager (VSM) in the [Cisco](#) 1000V or vCenter in the case of VMware's distributed switch. Open

vSwitch is meant to be controlled and managed by third party controllers and managers like OpenDaylight, POX Controller, etc. This doesn't make OVS dependable on a SDN controller at all instances. OVS can be deployed on all servers in an environment and operate with traditional MAC learning functionality [11][12].

#### 2.4.1 Open vSwitch Key Components

OVS mainly consists of the following components [12]:

- **ovs-vswitchd**, a component used for flow-based switching by implementing the virtual switch, along with a supporting Linux kernel module.
- **ovsdb-server**, a database server that holds ovs-vswitchd configuration.
- **ovs-dpctl**, a tool used for switch kernel module configuration.
- **ovs-vsctl**, a utility used for querying and updating the state of ovs-vswitchd in the ovsdb-server.
- **ovs-appctl**, a utility that sends commands to running ovs-vswitchd.

Open vSwitch also provides some tools:

- **ovs-ofctl**, a utility for monitoring and administering OpenFlow switches and controllers.
- **ovs-pki**, a utility for setting up and managing a public-key infrastructure for OpenFlow switches.
- **ovs-testcontroller**, a simple OpenFlow controller that manages multiple switches over the OpenFlow protocol.

## 2.5 Characteristics of Open vSwitch

Because of the need to allocate VMs of the same network on different physical servers belonging to different datacenters, hypervisors must now have the capability to bridge traffic between VMs across different servers. Open vSwitch is used here to help achieve layer 2 communication over layer 3. Open vSwitch is targeted at multi-server virtualization deployments involving dynamic end-points, maintenance of logical abstraction and sometimes integration with special purpose switching hardware, which L2 switching is not suited for [12].

Open vSwitch has the following characteristics which makes it a good fit to support multi-server virtualization environments [12]:

- Open vSwitch supports configuration and migration of both fast network state and soft state of instances (VMs). Soft state involves L2 learning table entries, L3 forwarding states, ACLs, QoS policies, configuration monitoring like Netflow, sFlow, IPFIX, etc. associated with the instance which are also necessary to be identified and migrated between physical servers easily. Open vSwitch not only helps migrate associated configurations but also any live network state or the existing state of the VMs which might be difficult to reconstruct otherwise.
- Open vSwitch mainly supports a network state database (OVSDB) which supports remote triggers. This is specifically useful in dynamic network scenarios where VMs are added and removed time-to-time and are moved back and forth in time with changes in logical network environment. Now any associated software can track various aspects of network and can respond as and when they change.

- In addition, Open vSwitch also supports OpenFlow, which allows us to deploy innovative routing and switching protocols in our network as it separates the data path, which still resides in the switch, and control path, which is transferred to the controller.
- Open vSwitch includes multiple methods for maintaining logical tags that is used to identify a VM belonging to a logical network. These tagging rules are accessible to the remote controller and not coupled with networking devices thus helping thousands of tagging and address remapping rules to be configured and migrated.
- Hardware integration of Open vSwitch to chipsets is possible as Open vSwitch's forwarding datapath is also designed to offload packet processing to hardware chipset, which allows it to not only control a software virtual switch but also a hardware switch.

All these features of Open vSwitch helps it to minimize in-kernel code as much as possible and reuse existing subsystem when required. Thus reducing the load off the kernel and delivering better performance with better scalability [12].

## Chapter 3

### VXLAN COMMUNICATION

Virtual machines belonging to the same network, residing on physical hosts belonging to different datacenters need to communicate via layer 3. VXLAN plays a major role in tunneling the source VM's layer 2 traffic over layer 3 to the host which has destination VM, without VMs knowledge. The VXLAN tunnel providing the overlay network handles transportation of these packets effectively connecting VMs on different host making them believe that they have a direct layer 2 connection to one another [2].

VXLAN tunneling uses the Open vSwitch which has port connections to the VMs running in the native host. The Open vSwitch connects to VXLAN gateway, called as the Virtual Tunnel End Points (VTEP), which is the key element that provides the encapsulation and de-encapsulation function. They are the point of entry for the VXLAN tunnel in the host hypervisor. All the encapsulation and de-encapsulation of the packet occur at this entry point and there is no requirement for any separate configuration to the VMs residing in the host hypervisor [4].

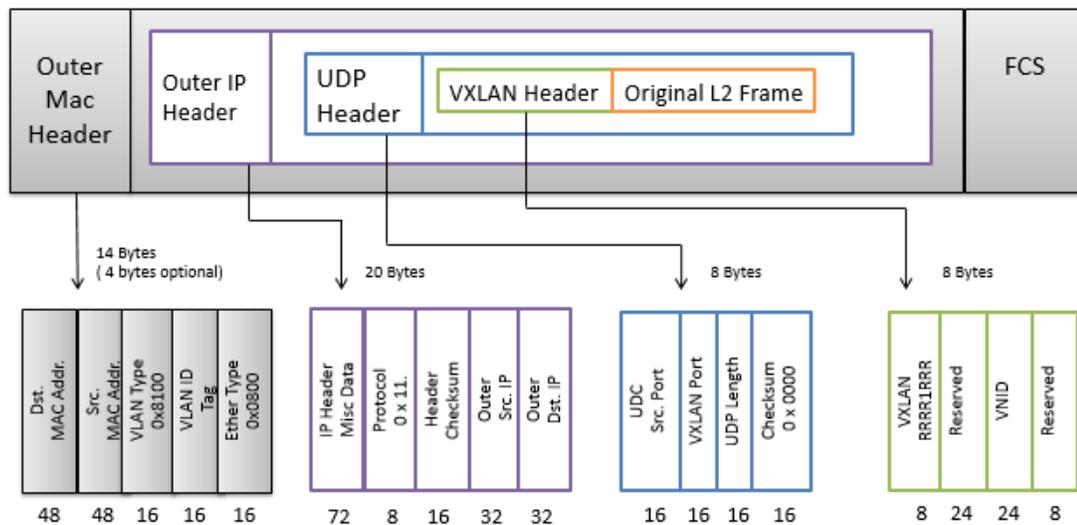
VTEP associated with the OVS switch does packet encapsulation based on the destination IP of the VM. If the virtual machine tries to communicate with another virtual machine residing in the same host, then the VTEP doesn't perform any kind of encapsulation and the traffic is switched locally [18].

If one VM tries to reach another VM which belongs to the same network but resides on a physical server in a different datacenter, for example, then all the traffic

coming from the client VM is encapsulated with appropriate VXLAN header and forwarded to the destination address referred in the OpenFlow table entry. VTEP on the other end after receiving this packet de-encapsulates this and delivers to the recipient VM. The VTEP decides to transfer the inner layer 2 packet frame to the correct VM based on the unique VXLAN Network Identifier (VNI), a 24-bit address space that helps scale virtual network in VXLAN more than the available 4096 VMs as in traditional 802.1q VLANs [4].

For a broadcast traffic like ARP request, the local VTEP encapsulates the packet with VXLAN header and multicasts the frame to all the hosts belonging to the VNI. The recipient VTEP which has the destination host belonging to the same VNI, de-encapsulates it and processes it as unicast traffic [3][4].

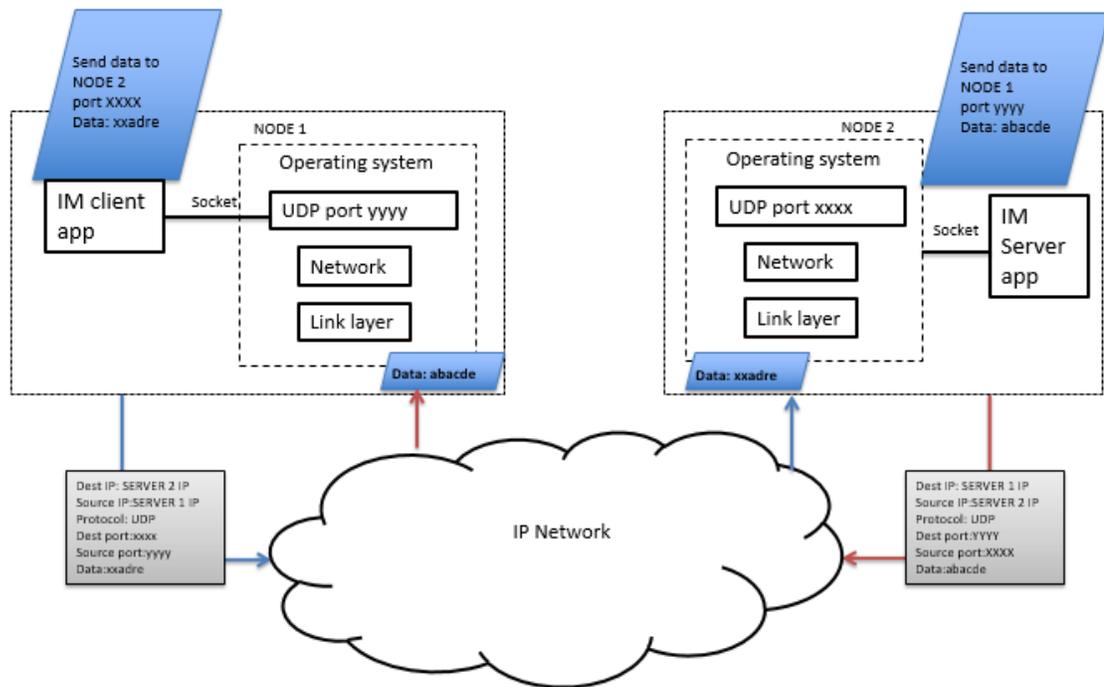
The VXLAN encapsulation of the packet leaving the switch to a destination host is described in figure 3.1.



**Figure 3.1:** VXLAN frame encapsulation (Image adapted from [3])

### 3.1 Traditional Client Server communication

In case of UDP transport layer multiplexing, the source node along with its IP address assigns a random source port for its packets and sends it to the destination IP and port “xxxx” at which the destination node is listening to for packets. On receiving the packets, the destination node sends out a reply with its IP and port “xxxx” as source address and the client node’s IP and port as the destination address. The destination node takes the IP and port number of the sending node from its packet header and uses it to send a reply to the node using it as the destination address (Refer figure 3.2).



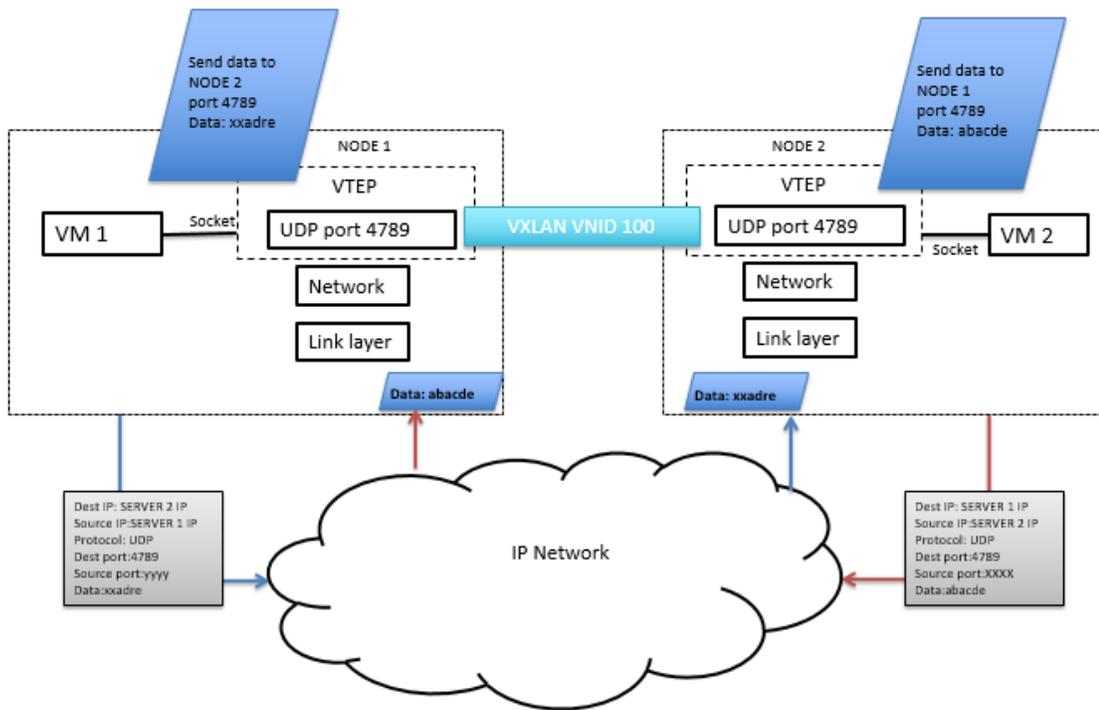
**Figure 3.2:** UDP communication

### **3.2 VXLAN Communication between VTEPs**

It is important to understand how a VXLAN communication differs from a usual UDP communication between two nodes. In VXLAN communication both the end nodes have a tunnel end point and have VXLAN listening port 4789 open for packets reception. All the packets transmitted from each VTEP is destined to the VXLAN destination port 4789 where the VTEP is listening to.

So, in a VXLAN bridged network, an UDP encapsulated packet from node 1 has a source IP of node 1 and a random port number assigned by the OS say “xxxx” and a destination IP of node 2 and port 4789. The node 2 replies with a UDP encapsulated packet with a source IP of node 2 and a random port number assigned by OS say “yyyy” and the destination IP of node 1 and port 4789 (refer figure 3.3).

Thus, the destination port number remains the same 4789 for all packets between node1 and node2. The sending and receiving port are never the same for a node which is connected to the VXLAN tunnel.



**Figure 3.3** VXLAN communication

## Chapter 4

### MININET

Mininet is a network emulation software which enables to launch a network of virtual host, switches, and SDN controller all with a single command. The host spun up by mininet run standard Linux network software. It supports OpenFlow for easy and flexible routing techniques and SDN implementations at lab environments [13].

Mininet was created in first place to support research, development, testing, debugging and other befitting tasks to perform on an experimental network in a laboratory environment to learn about SDN and OpenFlow. It uses lightweight virtualization to support this workflow. A user can implement any kind of network topology, configure routing mechanisms and control traffic rate on their test environments on a single laptop and can deploy their network in real production network. It utilizes minimal disk space to support the user virtual network, leveraging Linux features to launch hundreds of switches, hosts and controllers along with gigabits of bandwidth with a single command line or an API [13].

Mininet comes as a single Linux package and is compatible with hypervisor applications like VMware, VirtualBox, etc. Any user who wants to use mininet can download its .iso file to install and run it on their VirtualBox or can download the mininet application straight from their Linux distribution. For example, a user using Ubuntu can download mininet using the command:

```
sudo apt-get install mininet
```

The following attributes makes mininet an ideal prototyping workflow for users looking to build large virtual network with a constraint of a very limited resource [14]:

- Flexibility: mininet defines and supports new technologies and functionalities in its software using python libraries on a Linux package.
- Deployable: A functionally correct prototype in mininet can be deployed to a hardware-based network without any change of code or configuration.
- Interactive: mininet creates an interactive environment with the defined network such that it gives a feel of managing and running a real-time network.
- Realistic: mininet creates real behavior networks with high degree of confidence. There is no need for modification of any protocol stack or application for defining a network in mininet.
- Sharable: mininet projects can be shared with peers and collaborators where they can run and modify that experiment.

Mininet uses virtual Ethernet pair links between its hosts which is basically a shell process and the switch. Mininet uses OpenFlow switches and connects to a controller that resides in the host VM that mininet is running on. So mininet packages all these things inside to give an ease of work experience to the users who can simulate a network by a single command. For example, the command:

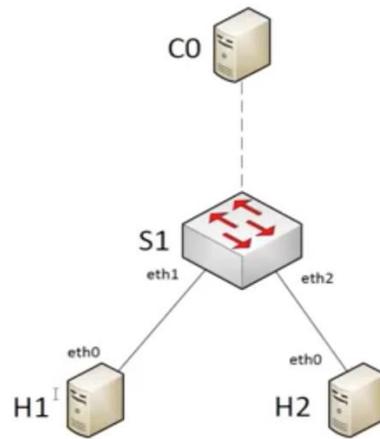
```
sudo mn --switch ovs --controller ref --topo tree,depth=2,fanout=8 --test pingall
```

spins up a network having a tree topology of depth 2 and fanout 8 (i.e. 64 hosts connected to 9 switches) using Open vSwitch controlled by a reference controller and then runs a pingall test to check the connectivity between these hosts [29].

We use mininet for the above-mentioned reasons to set up our overlay network and add VXLAN tunnel between our client and server.

#### **4.1 Setting up Mininet**

In our setup we use Mininet on each of the hosts to spin up our simple network topology, which consists of a Open vSwitch S1, and two hosts H1 and H2 connected to the switch (Refer figure 4.1). This switch is the Open vSwitch which will help us configure the VTEP port for VXLAN tunnel, and the host H1 and H2 are configured to belong to different VXLAN segment. So H1 and H2 though reside on the same host, can have same MAC address and local network IP, as both of their network are independent of each other. The idea is to keep the topology as simple as possible as we focus on setting up a VXLAN tunnel under a secured connection between the hosts and just require two VMs communicating over the tunnel to test our secured connection.

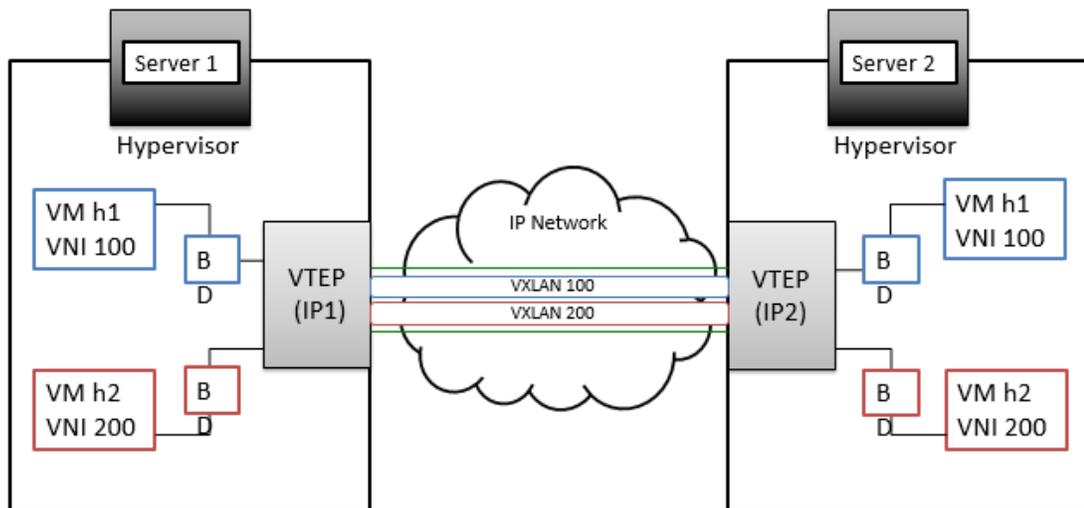


**Figure 4.1:** Mininet Topology (Image adapted from [29]).

## Chapter 5

### VXLAN TUNNEL CONFIGURATION ON OVERLAY NETWORK

The laboratory setup used for our study consists of two Linux hosts machines, Server1 and Server2 which represent physical servers each belonging to different datacenters, ensuring a requirement of a router to perform a layer 3 communication between the two hosts. This is to confer that the communication via the VXLAN tunnel occurs through the overlay network. The laboratory virtual network that we built is depicted in figure 5.1.



**Figure 5.1:** Overlay network set up using VXLAN

## 5.1 Mininet Configuration at Server1

Now a simple command `sudo mn` in mininet creates the default network topology with two hosts connected to a switch with a default controller [19].

```
root@ubuntu:~# mn
*** No default OpenFlow controller found for default switch!
*** Falling back to OVS Bridge
*** Creating network
*** Adding controller
*** Adding hosts:
h1 h2
*** Adding switches:
s1
*** Adding links:
(h1, s1) (h2, s1)
*** Configuring hosts
h1 h2
*** Starting controller
*** Starting 1 switches
s1 ...
*** Starting CLI:
```

**Figure 5.2:** Mininet set-up at server1

The information of all the nodes in the working topology can be viewed using the dump command. It can be verified that two tenant VMs and a switch are created (figure 5.3).

```
mininet> dump
<Host h1: h1-eth0:10.0.0.1 pid=8649>
<Host h2: h2-eth0:10.0.0.2 pid=8652>
<OVSBridge s1: lo:127.0.0.1,s1-eth1:None,s1-eth2:None pid=8658>
```

**Figure 5.3:** Dump command to view mininet topology components for server1

To execute a command on a host, the host name should be mentioned first on the mininet CLI followed by the command for that host.

Now for the ease of use, the IP address and the randomly assigned MAC address is configured for both VM h1 and h2 to convenient addresses.

### 5.1.1 Changing IP Configuration for node h1

The following commands were used to assign user specified addresses for node h1 (figure 5.4):

```
mininet> h1 ifconfig h1-eth0 10.0.0.1 netmask 255.0.0.0
```

```
mininet> h1 ifconfig h1-eth0 hw ether 00:00:00:00:00:01
```

```
mininet> h1 ifconfig
h1-eth0  Link encap:Ethernet  HWaddr 00:00:00:00:00:01
         inet addr:10.0.0.1  Bcast:10.255.255.255  Mask:255.0.0.0
         inet6 addr: fe80::1ccf:31ff:feae:ebdb/64 Scope:Link
         UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
         RX packets:33 errors:0 dropped:0 overruns:0 frame:0
         TX packets:12 errors:0 dropped:0 overruns:0 carrier:0
         collisions:0 txqueuelen:1000
         RX bytes:3610 (3.6 KB)  TX bytes:928 (928.0 B)

lo       Link encap:Local Loopback
         inet addr:127.0.0.1  Mask:255.0.0.0
         inet6 addr: ::1/128 Scope:Host
         UP LOOPBACK RUNNING  MTU:65536  Metric:1
         RX packets:0 errors:0 dropped:0 overruns:0 frame:0
         TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
         collisions:0 txqueuelen:1
         RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

**Figure 5.4:** VM1 Ethernet configuration at server1

### 5.1.2 Changing IP Configuration for node h2

The following commands were used to assign user specified addresses for node h2 (figure 5.5):

```
mininet> h2 ifconfig h2-eth0 10.0.0.2 netmask 255.0.0.0
```

```
mininet> h2 ifconfig h2-eth0 hw ether 00:00:00:00:00:02
```

```
mininet> h2 ifconfig
h2-eth0  Link encap:Ethernet  HWaddr 00:00:00:00:00:02
         inet addr:10.0.0.2  Bcast:10.255.255.255  Mask:255.0.0.0
         inet6 addr: fe80::1042:aff:feef:28d2/64 Scope:Link
         UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
         RX packets:51 errors:0 dropped:0 overruns:0 frame:0
         TX packets:8 errors:0 dropped:0 overruns:0 carrier:0
         collisions:0 txqueuelen:1000
         RX bytes:6776 (6.7 KB)  TX bytes:648 (648.0 B)

lo       Link encap:Local Loopback
         inet addr:127.0.0.1  Mask:255.0.0.0
         inet6 addr: ::1/128 Scope:Host
         UP LOOPBACK RUNNING  MTU:65536  Metric:1
         RX packets:0 errors:0 dropped:0 overruns:0 frame:0
         TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
         collisions:0 txqueuelen:1
         RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

**Figure 5.5** VM2 Ethernet configuration at server1

The correct configuration of these VMs can be verified by pinging from each host to its own modified IP.

### 5.1.3 VXLAN tunnel set-up

Open vSwitch with bridge s1 and its associated ports as per the mininet topology can be viewed using the command as seen in figure 5.6:

```

root@ubuntu:~# ovs-vsctl show
feaf9ab0-03fe-4de7-a1fe-400ab10cfdca
Bridge "s1"
  Controller "ptcp:6634"
  fail_mode: standalone
  Port "s1-eth2"
    Interface "s1-eth2"
  Port "s1-eth1"
    Interface "s1-eth1"
  Port "s1"
    Interface "s1"
      type: internal
  ovs_version: "2.5.0"

```

**Figure 5.6:** OVS bridge and port details at server1

Figure 5.6 shows that s1 has ports “s1-eth1” and “s1-eth2” which are connected to the two hosts h1 and h2 respectively. Each of these ports has an interface with the same name. The difference between a port and an interface is that a port can contain multiple interfaces. You can use a port to create multiple bonds for like LACP aggregation. Here since we are not doing any multiple bonds, we see just one interface associated with each port.

Next the VXLAN tunnel is established between the two switches. For this bridge s1 should be configured to add the Virtual Tunnel End Point (VTEP) to it using the following command:

```

root@ubuntu:~# ovs-vsctl add-port s1 vtep -- set interface vtep type=vxlan
option:remote_ip=128.4.13.229 option:key=flow ofport_request=10

```

Breakdown of the command to understand what it does:

**add-port s1 vtep :** adds a port named “vtep” to the switch s1 (the name of the vtep port need not necessarily be vtep and can be anything).

**-- set interface vtep type=vxlan** : sets the interface vtep of type vxlan. If a user is creating a GRE tunnel then they can mention it as type=gre

**option:remote\_ip=128.4.13.229** : Remote IP is the IP address of the other server (server2) that this host (server1) is to be connected via the tunnel.

**option:key=flow** : This part is to specify the VNI number to identify each logical tenant traffic. We give *key=flow* to provide overloading of the tunnel command, where the packet flow through the VTEP is directed by the OpenFlow flow entries. We assign these OpenFlow entries to the Open vSwitch later.

**ofport\_request=10** : This part specifies that we want to use OpenFlow port 10 for this VTEP port. Without this we wouldn't know which OpenFlow port number is allocated and would be a problem for the OpenFlow flow entries later.

Verifying from figure 5.7, the port *vtep* is now added to the switch *s1* with type as *vxlan* and the *key* and *remote\_ip* options as specified.

```
root@ubuntu:~# ovs-vsctl show
feaf9ab0-03fe-4de7-a1fe-400ab10cfdca
  Bridge "s1"
    Controller "ptcp:6634"
    fail_mode: standalone
    Port "s1-eth2"
      Interface "s1-eth2"
    Port "s1-eth1"
      Interface "s1-eth1"
    Port vtep
      Interface vtep
        type: vxlan
        options: {key=flow, remote_ip="128.4.13.229"}
    Port "s1"
      Interface "s1"
        type: internal
  ovs_version: "2.5.0"
```

**Figure 5.7:** Addition of VTEP to OVS bridge at server1

Next, we can confirm that the port name VTEP is mapped to the OpenFlow port 10 as requested using the command (seen in figure5.8):

```
# ovs-ofctl show s1
```

```
root@ubuntu:~# ovs-ofctl show s1
OFPT_FEATURES_REPLY (xid=0x2): dpid:0000000000000001
n_tables:254, n_buffers:256
capabilities: FLOW_STATS TABLE_STATS PORT_STATS QUEUE_STATS ARP_MATCH_IP
actions: output enqueue set_vlan_vid set_vlan_pcp strip_vlan mod_dl_src mod_dl_dst mod_
nw_src mod_nw_dst mod_nw_tos mod_tp_src mod_tp_dst
1(s1-eth1): addr:86:95:d4:26:a1:ba
  config:      0
  state:      0
  current:    10GB-FD COPPER
  speed: 10000 Mbps now, 0 Mbps max
2(s1-eth2): addr:e6:b9:c8:f2:04:94
  config:      0
  state:      0
  current:    10GB-FD COPPER
  speed: 10000 Mbps now, 0 Mbps max
10(vtep): addr:96:1b:f3:93:75:b9
  config:      0
  state:      0
  speed: 0 Mbps now, 0 Mbps max
LOCAL(s1): addr:ba:a9:04:45:7c:4c
  config:      PORT_DOWN
  state:      LINK_DOWN
  speed: 0 Mbps now, 0 Mbps max
OFPT_GET_CONFIG_REPLY (xid=0x4): frags=normal miss send len=0
```

**Figure 5.8:** OpenFlow port details of bridge s1 at server1

#### 5.1.4 Addition of Flow table entry

The next step is to add flow entries to direct traffic between the tenants that belong to the same network.

Note that the VTEPs need to have a table mapping VM MAC addresses to virtual network end point IP addresses. There are different ways by which the VTEPs can learn these mappings [10]:

- **Manual :** Configuration include manual flow entries. These are simple and more suited for lab learning environment.

- **Push process** : VTEPs can also learn MAC address mapping through a push process like using a SDN controller where the controller takes care of the traffic forwarding by pushing flow entries to the switch.
- **Pull process** : With pull process, the VTEPs can request flow entries from a central directory.

We use the manual method in our laboratory setup for our VTEPs to control VM traffic from the switch, as this is a simple network between two servers and the key idea is to transfer encapsulated packets from server1 to server2 through the tunnel. There is no need for a controller to control this simple traffic flow.

#### 5.1.4.1 Flow entries at Server1

Our flow table entry on server 1 consists of two tables Table 0 and Table 1 as shown in figure 5.9 and figure 5.10. Table 0 is used to tag flows with the VXLAN VNIs and Table 1 is used to forward traffic.

Table 0:

```
table=0,in_port=1,actions=set_field:100->tun_id,redirect(,1)
table=0,in_port=2,actions=set_field:200->tun_id,redirect(,1)
table=0,actions=redirect(,1)
```

**Figure 5.9:** Flow table 0 for server1

All the packets which are coming in from port 1 are assigned to have a *tun\_id* of 100, i.e. all the packets from VM h1 which is connected to port1 get a VNI number of 100 and *redirect(,1)* asks to move to table 1 after the action is taken. Similarly, for

all the packets coming in from port 2 of the switch s1, VNI number 200 is assigned and the control is then given to table1.

The last entry is the default entry. When there is no match for flow entries, its default action is taken which is to move to Table1.

Table 1:

```
table=1,tun_id=100,dl_dst=00:00:00:00:00:01,actions=output:1
table=1,tun_id=200,dl_dst=00:00:00:00:00:02,actions=output:2
table=1,tun_id=100,dl_dst=00:00:00:00:00:03,actions=output:10
table=1,tun_id=200,dl_dst=00:00:00:00:00:04,actions=output:10
table=1,tun_id=100,arp,nw_dst=10.0.0.1,actions=output:1
table=1,tun_id=200,arp,nw_dst=10.0.0.2,actions=output:2
table=1,tun_id=100,arp,nw_dst=10.0.0.3,actions=output:10
table=1,tun_id=200,arp,nw_dst=10.0.0.4,actions=output:10
table=1,priority=100,actions=drop
```

**Figure 5.10:** Flow table 1 for server1

Table 1 forwards packets to the corresponding output ports depending on the flow match condition as follows in order of the flow entries mentioned in figure 5.10:

- If the incoming packet has a tun\_id (VNI) =100 and the destination MAC address=00:00:00:00:00:01, then forward it through the port 1.
- If the incoming packet has a tun\_id (VNI) =200 and the destination MAC address=00:00:00:00:00:02, then forward it through the port 2.
- If the incoming packet has a tun\_id (VNI) =100 and the destination MAC address=00:00:00:00:00:03, then forward it through the port 10.
- If the incoming packet has a tun\_id (VNI) =200 and the destination MAC address=00:00:00:00:00:04, then forward it through the port 10.

- If the incoming packet has a tun\_id (VNI) =100 and it's an ARP message with the destination IP address=10.0.0.1, then forward it through the port 1.
- If the incoming packet has a tun\_id (VNI) =200 and it's an ARP message with the destination IP address=10.0.0.2, then forward it through the port 2.
- If the incoming packet has a tun\_id (VNI) =100 and it's an ARP message with the destination IP address=10.0.0.3, then forward it through the port 10.
- If the incoming packet has a tun\_id (VNI) =200 and it's an ARP message with the destination IP address=10.0.0.4, then forward it through the port 10.
- Default condition is to assign unmatched packets with a priority of 100 and is to be dropped.

For the ease of applying these flow entries to the switch s1, the flow table entries are put in a file called flowtable.txt and then appended to the OVSdb using the command:

```
root@ubuntu:~# ovs-ofctl add-flows s1 flowtable.txt
```

The flow entries in the OVSdb can be verified by the OVS dump flow command which will dump all the flow entries stored in the OVS database:

```
root@ubuntu:~# ovs-ofctl dump-flows s1
```

## 5.2 Mininet Configuration at Server2

Similar to server1, a default mininet topology is created using the command `sudo mn` as shown in figure 5.11:

```
user2@server2:~$ sudo mn
[sudo] password for user2:
*** No default OpenFlow controller found for default switch!
*** Falling back to OVS Bridge
*** Creating network
*** Adding controller
*** Adding hosts:
h1 h2
*** Adding switches:
s1
*** Adding links:
(h1, s1) (h2, s1)
*** Configuring hosts
h1 h2
*** Starting controller
*** Starting 1 switches
s1 ...
*** Starting CLI:
```

**Figure 5.11:** Mininet set-up at server2

The information of all the nodes in the working topology can be viewed using the `dump` command to verify that two tenant VMs and a switch are created.

```
mininet> dump
<Host h1: h1-eth0:10.0.0.1 pid=29551>
<Host h2: h2-eth0:10.0.0.2 pid=29554>
<OVSBridge s1: lo:127.0.0.1,s1-eth1:None,s1-eth2:None pid=29560>
```

**Figure 5.12:** Dump command to view mininet topology components for server2

To execute a command on a host, the host name is mentioned first on the mininet CLI followed by the command for that host.

The IP address and the randomly assigned MAC address for VM h1 and h2 are modified to our convenient addresses, so that packet follow up is made easy.

### 5.2.1 Changing IP Configuration for node h1

The following commands were used to assign user specified addresses for node h1 (figure 5.13):

```
mininet> h1 ifconfig h1-eth0 10.0.0.3 netmask 255.0.0.0
```

```
mininet> h1 ifconfig h1-eth0 hw ether 00:00:00:00:00:03
```

```
mininet> h1 ifconfig
h1-eth0  Link encap:Ethernet  HWaddr 00:00:00:00:00:03
         inet addr:10.0.0.3  Bcast:10.255.255.255  Mask:255.0.0.0
         inet6 addr: fe80::a8be:6cff:fe9e:e613/64 Scope:Link
         UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
         RX packets:1983 errors:0 dropped:0 overruns:0 frame:0
         TX packets:722 errors:0 dropped:0 overruns:0 carrier:0
         collisions:0 txqueuelen:1000
         RX bytes:207223 (207.2 KB)  TX bytes:51396 (51.3 KB)

lo       Link encap:Local Loopback
         inet addr:127.0.0.1  Mask:255.0.0.0
         inet6 addr: ::1/128 Scope:Host
         UP LOOPBACK RUNNING  MTU:65536  Metric:1
         RX packets:180 errors:0 dropped:0 overruns:0 frame:0
         TX packets:180 errors:0 dropped:0 overruns:0 carrier:0
         collisions:0 txqueuelen:1
         RX bytes:18441 (18.4 KB)  TX bytes:18441 (18.4 KB)
```

**Figure 5.13:** VM1 Ethernet configuration at server2

## 5.2.2 Changing IP Configuration for node h2

The following commands were used to assign user specified addresses for node h2 (figure 5.14):

```
mininet> h2 ifconfig h2-eth0 10.0.0.4 netmask 255.0.0.0
```

```
mininet> h2 ifconfig h2-eth0 hw ether 00:00:00:00:00:04
```

```
mininet> h2 ifconfig
h2-eth0  Link encap:Ethernet  HWaddr 00:00:00:00:00:04
         inet addr:10.0.0.4  Bcast:10.255.255.255  Mask:255.0.0.0
         inet6 addr: fe80::b09e:5ff:feb3:82de/64 Scope:Link
         UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
         RX packets:2290 errors:0 dropped:0 overruns:0 frame:0
         TX packets:117 errors:0 dropped:0 overruns:0 carrier:0
         collisions:0 txqueuelen:1000
         RX bytes:235518 (235.5 KB)  TX bytes:8362 (8.3 KB)

lo       Link encap:Local Loopback
         inet addr:127.0.0.1  Mask:255.0.0.0
         inet6 addr: ::1/128 Scope:Host
         UP LOOPBACK RUNNING  MTU:65536  Metric:1
         RX packets:25 errors:0 dropped:0 overruns:0 frame:0
         TX packets:25 errors:0 dropped:0 overruns:0 carrier:0
         collisions:0 txqueuelen:1
         RX bytes:2408 (2.4 KB)  TX bytes:2408 (2.4 KB)
```

**Figure 5.14:** VM2 Ethernet configuration at server2

The configuration of these VMs can be verified by pinging from each host to its own new modified IP.

### 5.2.3 VXLAN tunnel set-up

Open vSwitch with bridge s1 and its associated ports as per the mininet topology can be viewed using the command as seen in the figure 5.15:

```
root@server2:~# ovs-vsctl show
044a8189-b46d-470e-a863-c43745868978
  Bridge "s1"
    Controller "ptcp:6634"
    fail_mode: standalone
    Port "s1-eth2"
      Interface "s1-eth2"
    Port "s1-eth1"
      Interface "s1-eth1"
    Port "s1"
      Interface "s1"
        type: internal
```

**Figure 5.15:** OVS bridge and port details at server2

Next the VXLAN tunnel between the two switches is configured. For this bridge s1 is configured to add the VTEP to it using the command:

```
root@server2:~# ovs-vsctl add-port s1 vtep -- set interface vtep type=vxlan
option:remote_ip=128.4.95.66 option:key=flow ofport_request=10
```

Verifying from figure 5.16, the port *vtep* is now added to the switch *s1* with type as *vxlan* and the *key* and *remote\_ip* options as specified.

```
root@server2:~# ovs-vsctl show
044a8189-b46d-470e-a863-c43745868978
  Bridge "s1"
    Controller "ptcp:6634"
    fail_mode: standalone
    Port "s1-eth2"
      Interface "s1-eth2"
    Port "s1-eth1"
      Interface "s1-eth1"
    Port "s1"
      Interface "s1"
        type: internal
    Port vtep
      Interface vtep
        type: vxlan
        options: {key=flow, remote_ip="128.4.95.66"}
  ovs_version: "2.5.0"
```

**Figure 5.16:** Addition of VTEP to OVS bridge at server2

Next, we can confirm that the port name VTEP is mapped to the OpenFlow port 10 as requested using the command (seen in figure5.17):

```
# ovs-ofctl show s1
```

```
root@server2:~# ovs-ofctl show s1
OFPT_FEATURES_REPLY (xid=0x2): dpid:0000000000000001
n_tables:254, n_buffers:256
capabilities: FLOW_STATS TABLE_STATS PORT_STATS QUEUE_STATS ARP_MATCH_IP
actions: output enqueue set_vlan_vid set_vlan_pcp strip_vlan mod_dl_src mod_dl_d
st mod_nw_src mod_nw_dst mod_nw_tos mod_tp_src mod_tp_dst
1(s1-eth1): addr:ba:87:e7:50:e5:d2
  config: 0
  state: 0
  current: 10GB-FD COPPER
  speed: 10000 Mbps now, 0 Mbps max
2(s1-eth2): addr:26:cb:6f:6a:e7:fd
  config: 0
  state: 0
  current: 10GB-FD COPPER
  speed: 10000 Mbps now, 0 Mbps max
10(vtep): addr:16:80:ca:b1:93:0e
  config: 0
  state: 0
  speed: 0 Mbps now, 0 Mbps max
LOCAL(s1): addr:16:d6:0a:8e:e2:4b
  config: PORT_DOWN
  state: LINK_DOWN
  speed: 0 Mbps now, 0 Mbps max
OFPT_GET_CONFIG_REPLY (xid=0x4): frags=normal miss_send_len=0
```

**Figure 5.17:** OpenFlow port details of bridge s1 at server2

## 5.2.4 Addition of Flow table entry

The next step is to add flow entries to direct traffic between the tenants that belong to the same network.

### 5.2.4.1 Flow entry at Server2

The flow table entry table 0 and table 1 is used for the Open vSwitch on server2, similar to server1 with a difference of the destination IP and MAC addresses for the flow rules.

Table 0:

```
table=0,in_port=1,actions=set_field:100->tun_id,resubmit(,1)
table=0,in_port=2,actions=set_field:200->tun_id,resubmit(,1)
table=0,actions=resubmit(,1)
```

**Figure 5.18:** Flow table 0 for server2

The table 0 in figure 5.18 is the same as table 0 of server1, where we assign VNI number 100 to all the packets flowing in from port 1 of the switch s1 and a VNI number of 200 to all packets flowing in from port 2 of the switch s1, during UDP encapsulation.

The last entry is the default entry. Where if there is no match for flow entries, this default action is taken which is to move to Table1.

Table 1:

```
table=1,tun_id=100,dl_dst=00:00:00:00:00:03,actions=output:1
table=1,tun_id=200,dl_dst=00:00:00:00:00:04,actions=output:2
table=1,tun_id=100,dl_dst=00:00:00:00:00:01,actions=output:10
table=1,tun_id=200,dl_dst=00:00:00:00:00:02,actions=output:10
table=1,tun_id=100,arp,nw_dst=10.0.0.3,actions=output:1
table=1,tun_id=200,arp,nw_dst=10.0.0.4,actions=output:2
table=1,tun_id=100,arp,nw_dst=10.0.0.1,actions=output:10
table=1,tun_id=200,arp,nw_dst=10.0.0.2,actions=output:10
table=1,priority=100,actions=drop
```

**Figure 5.19:** Flow table 1 for server2

The table 1 forwards packets to the corresponding output ports depending on the flow match condition as follows in order of the flow entries mentioned in figure 5.19 :

- If the incoming packet has a tun\_id (VNI) =100 and the destination MAC address=00:00:00:00:00:03, then forward it through the port 1.
- If the incoming packet has a tun\_id (VNI) =200 and the destination MAC address=00:00:00:00:00:04, then forward it through the port 2.
- If the incoming packet has a tun\_id (VNI) =100 and the destination MAC address=00:00:00:00:00:01, then forward it through the port 10.
- If the incoming packet has a tun\_id (VNI) =200 and the destination MAC address=00:00:00:00:00:02, then forward it through the port 10.
- If the incoming packet has a tun\_id (VNI) =100 and it's an ARP message with the destination IP address=10.0.0.3, then forward it through the port 1.
- If the incoming packet has a tun\_id (VNI) =200 and it's an ARP message with the destination IP address=10.0.0.4, then forward it through the port 2.
- If the incoming packet has a tun\_id (VNI) =100 and it's an ARP message with the destination IP address=10.0.0.1, then forward it through the port 10.
- If the incoming packet has a tun\_id (VNI) =200 and it's an ARP message with the destination IP address=10.0.0.2, then forward it through the port 10.
- Default condition is to assign unmatched packets with a priority of 100 and is to be dropped.

For the ease of applying these flow entries to the switch s1, we put them in a file called flowtable.txt and append it to the OVSdb using the command:

```
root@server2:~# ovs-ofctl add-flows s1 flowtable.txt
```

### 5.3 Testing VXLAN tunnel by sending ICMP

Now since both server1 and server2 are connected via VXLAN tunnel, VMs having same VNI residing in different servers can communicate with each other through this VXLAN based overlay network as though they are having a layer 2 communication.

#### 5.3.1 ICMP from 10.0.0.1 and 10.0.0.2 at Server1

Figure 5.20 shows a successful ping communication from VM 10.0.0.1 to VM 10.0.0.3 where both VMs belong to VNI 100 and from VM 10.0.0.2 to VM 10.0.0.4 where both VMs belong to VNI 200.

```
mininet> h1 ping 10.0.0.3
PING 10.0.0.3 (10.0.0.3) 56(84) bytes of data.
64 bytes from 10.0.0.3: icmp_seq=1 ttl=64 time=17.3 ms
64 bytes from 10.0.0.3: icmp_seq=2 ttl=64 time=11.4 ms
64 bytes from 10.0.0.3: icmp_seq=3 ttl=64 time=12.5 ms
64 bytes from 10.0.0.3: icmp_seq=4 ttl=64 time=14.9 ms
64 bytes from 10.0.0.3: icmp_seq=5 ttl=64 time=11.7 ms
^C
--- 10.0.0.3 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4007ms
rtt min/avg/max/mdev = 11.446/13.588/17.337/2.244 ms
mininet> h2 ping 10.0.0.4
PING 10.0.0.4 (10.0.0.4) 56(84) bytes of data.
64 bytes from 10.0.0.4: icmp_seq=1 ttl=64 time=18.6 ms
64 bytes from 10.0.0.4: icmp_seq=2 ttl=64 time=14.7 ms
64 bytes from 10.0.0.4: icmp_seq=3 ttl=64 time=12.1 ms
64 bytes from 10.0.0.4: icmp_seq=4 ttl=64 time=14.0 ms
^C
--- 10.0.0.4 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3007ms
rtt min/avg/max/mdev = 12.166/14.911/18.691/2.379 ms
```

**Figure 5.20:** ICMP test from server1

### 5.3.2 ICMP from 10.0.0.3 and 10.0.0.4 at Server2

Figure 5.21 shows a successful ping communication from VM 10.0.0.3 to VM 10.0.0.1 where both VMs belong to VNI 100 and from VM 10.0.0.4 to VM 10.0.0.2 where both VMs belong to VNI 200.

```
mininet> h1 ping 10.0.0.1
PING 10.0.0.1 (10.0.0.1) 56(84) bytes of data.
64 bytes from 10.0.0.1: icmp_seq=1 ttl=64 time=16.6 ms
64 bytes from 10.0.0.1: icmp_seq=2 ttl=64 time=12.1 ms
64 bytes from 10.0.0.1: icmp_seq=3 ttl=64 time=11.7 ms
^C
--- 10.0.0.1 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2003ms
rtt min/avg/max/mdev = 11.728/13.514/16.647/2.224 ms
mininet> h2 ping 10.0.0.2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=14.2 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=13.2 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=11.8 ms
^C
--- 10.0.0.2 ping statistics ---
4 packets transmitted, 3 received, 25% packet loss, time 3004ms
rtt min/avg/max/mdev = 11.878/13.107/14.211/0.965 ms
```

**Figure 5.21:** ICMP test from server2

## Chapter 6

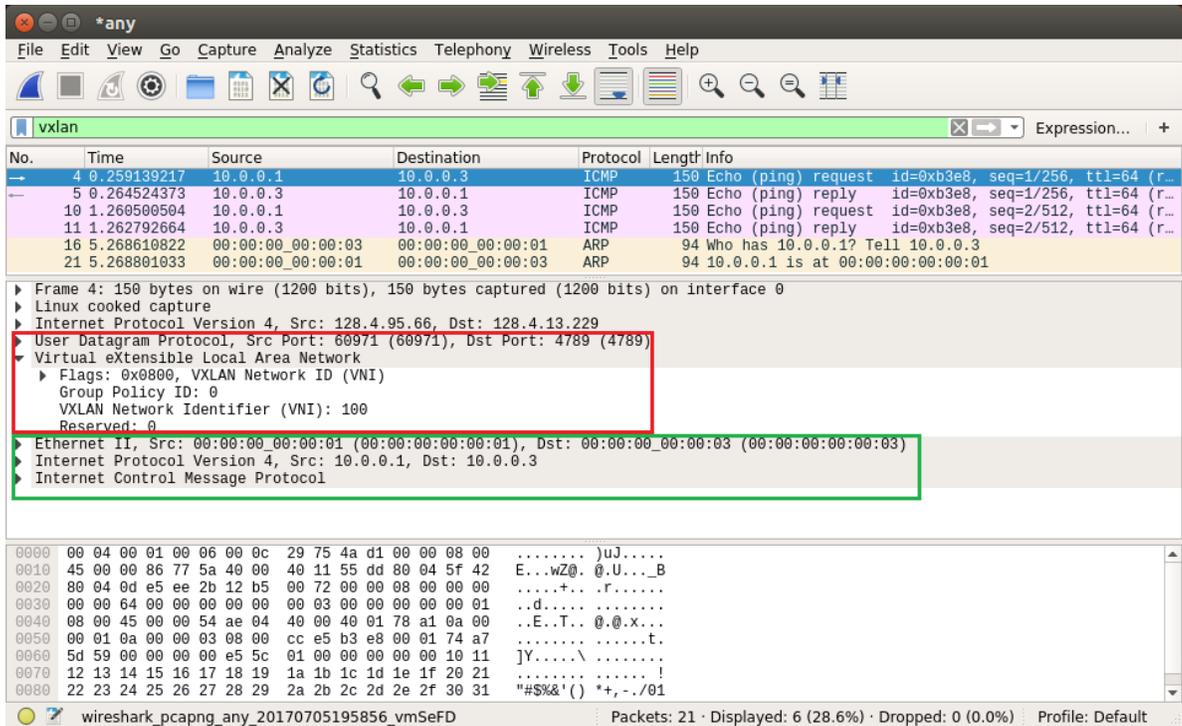
### STUDY OF PACKET FLOW THROUGH UNENCRYPTED VXLAN TUNNEL

#### 6.1 Wireshark packet capture

On capturing the packets on Wireshark, we would see UDP packets flowing out of the Ethernet having source address of server1 and destination address of server2 with port 4789. This is the UDP encapsulated packet by the server1's VTEP. The Wireshark doesn't know that these are VXLAN packets and thus doesn't show details of the layer2 payload being tunneled. To add VXLAN packet type to Wireshark:

- Right click on one of the packets and select "Decode As".
- In the Decode As dialogue box, add Field as UDP port and value as 4789 which is the default VXLAN listening port for our tunnel.
- Finally set the Current Field to VXLAN and click OK.

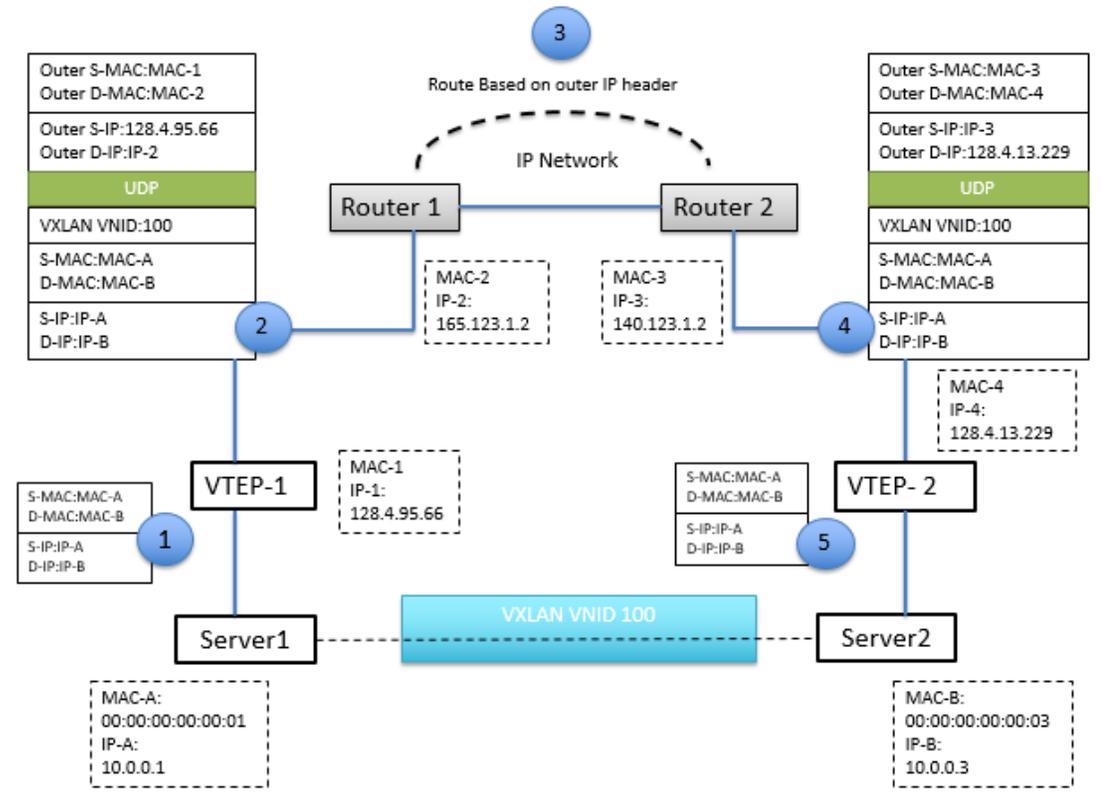
Once decoded Wireshark will now display the VXLAN headers (marked by red box in figure 6.1) along with the underneath layer 2 packet that has been encapsulated by the VTEP (marked by green box in figure 6.1).



**Figure 6.1:** Packet capture at unsecured VXLAN tunnel

## 6.2 Packet flow Walkthrough

To understand the flow of an ICMP packet between two VMs belonging to the same network, we consider the communication between VM h1 with IP address 10.0.0.1 and MAC address 00:00:00:00:00:01 at the server1 (128.4.95.66) and VM h3 with IP address 10.0.0.3 and MAC address 00:00:00:00:00:03 at the server2 (128.4.13.229). Note here server1 and server2 belong to different subnet.



**Figure 6.2:** Packet flow in unsecured VXLAN tunnel (Image adapted from [3])

With reference to figure 6.2, the packet flow of VXLAN packet can be explained step by step as following:

- At server1 (128.4.95.66), the ICMP packet from h1 with source MAC 00:00:00:00:00:01 and destination MAC 00:00:00:00:00:03 reaches VTEP.
- The VTEP at server1 on seeing the destination MAC address of the ICMP packet, acts on the packet based on the flow table entries. In our case all packets to MAC address 00:00:00:00:00:03 should be forwarded to server2 IP address. So the VTEP encapsulates this layer 2 ICMP packet with a UDP header with source IP 128.4.95.66 and a random source port 60971 assigned by

the OS and the destination IP 128.4.13.229 and destination port 4789 with the VNI number 100 to which both h1 and h3 belong.

- The layer 2 packet gets transmitted over layer 3 to reach server2 in which VM h3 resides.
- At server2 (128.4.13.229), the VTEP gateway at port 4789 receives the incoming VXLAN UDP packets and de-encapsulates them to get the underneath layer 2 ICMP packet. Depending on the VNI number in the UDP header (100 in this case), the VTEP at server2 forwards the de-encapsulated layer 2 packet to the correct VM 10.0.0.3
- The VM 10.0.0.3 on receiving the ICMP message from 10.0.0.1, sends out an ICMP reply to VM with source MAC 00:00:00:00:00:03 and destination MAC 00:00:00:00:00:01.
- The VTEP at server2 on receiving this layer 2 packet encapsulates it with a UDP header with source IP 128.4.13.229 and a random source port 57641 assigned by the OS and destination IP 128.4.95.66 and destination port 4789 with the VNI number 100 as per the flow table entry.
- This layer 2 ICMP reply then gets transmitted over layer 3 to reach server1.
- At server1 (128.4.95.66), the VTEP gateway at port 4789 receives this incoming VXLAN UDP packet and de-encapsulates the packet to get the underneath layer 2 ICMP reply. Looking at the VNI number in the UDP header, the VTEP at server1 forwards the layer 2 packet to the correct VM 10.0.0.1

## Chapter 7

### VXLAN TUNNEL CONFIGURATION WITH SSH

#### 7.1 Need for Tunnel Security

VXLAN does not define security on the overlay. There is no mechanism defined in IETF draft [2] to provide Confidentiality, Integrity and Authenticity (CIA) for VXLAN packets. This lack of security is also there for a layer 2 communication. But in case of Ethernet, for an attack to be injected, the attacker must be attached to the data link. This attack surface is minimal here as the only potential threat would be from within the datacenter and the network within the datacenter is usually physically secure. Traditional layer 2 attacks can also be mitigated by limiting the management and administrative scope of deploying and managing VMs in a VXLAN environment [16].

This attack surface is significantly increased on utilizing a MAC-in-IP scheme like VXLAN. In case of a typical VXLAN packet flow scenario as per figure 6.1, we see that the UDP packets over layer 3 are unencrypted and the underneath layer 2 packet can be viewed by anyone using packet capture application like Wireshark. Since the VTEP on the network are accessible by IP, traffic can be directed towards them with an inner Ethernet frame as though a legitimate server in the datacenter sent the packet thus allowing the attacker to impersonate that server and gain control over the server communication.

Open access to VXLAN UDP unencrypted packets can possibly lead to all kinds of attacks which can compromise the integrity and authenticity of the VXLAN tunnel communication [25].

Our target is to make this VXLAN overlay communication secure across datacenters, using any of the easy encryption methods that is commonly used and can be implemented in a lab environment. We chose SSH tunnel encapsulation technique for this purpose to see if it can successfully encapsulate VXLAN packets leaving the VTEP.

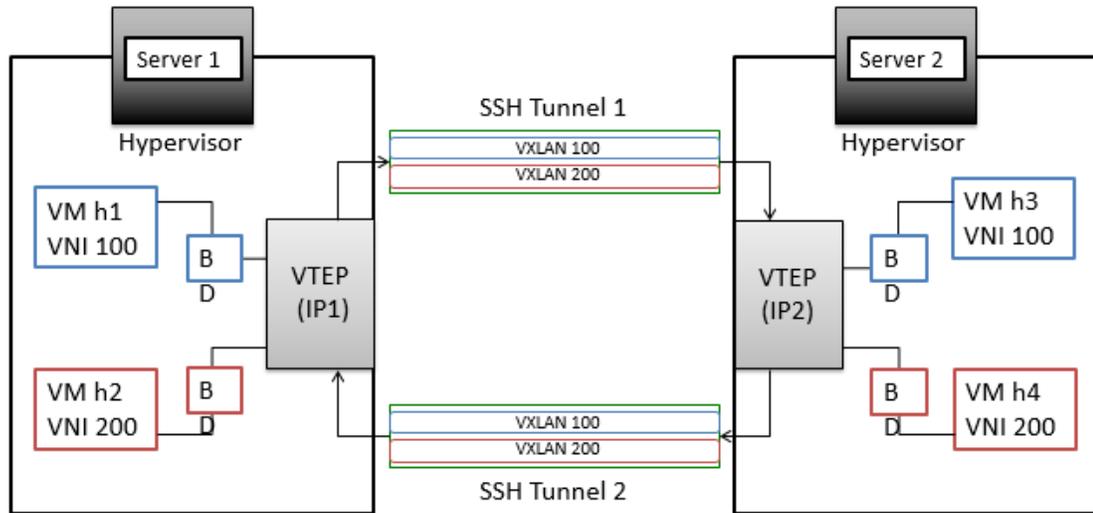
VXLAN is a tunneling communication technique between hosts for a layer 2 over layer 3. Creating a secure SSH tunnel over this tunnel could be challenging given the fact that SSH works by default on TCP ports and VXLAN is UDP based. Also considering the UDP encapsulation method and the de-encapsulation behavior of the VTEP, certain aspects of packet forwarding mechanism might have to be configured to make the communication compatible with SSH.

## **7.2 Network topology of VXLAN tunnel through SSH**

Adding a single SSH network between both the servers wouldn't result in a secured communication as in usual case. As we know that VXLAN tunnel uses separate source and destination ports at each node, we need to create two SSH networks between server1 and server2, one for packets from server1 VTEP destined to server2 VTEP and the other SSH tunnel for packets originating from server2 VTEP and destined to server1 VTEP.

To add a SSH tunnel over the VXLAN tunnel, we install OpenSSH client and OpenSSH server on both server1 and server2 having VMs h1 and h3 respectively as

both servers create a SSH tunnel for packets originating at their end. The figure 7.1 depicts the network topology used in this study for VXLAN tunnel through SSH.



**Figure 7.1:** Network topology of VXLAN tunnel through SSH

### 7.3 VXLAN Configuration set-up with SSH

SSH by default is configured to run over TCP and it encrypts and forwards TCP packets. Since VXLAN tunnel does UDP encapsulation of layer 2 packets, we must tunnel VXLAN UDP packets through SSH tunnel using UDP to TCP forwarding mechanism [17]. The following are the step by step procedure we follow to do so:

#### 7.3.1 For SSH tunnel 1 from Server1 to Server2

##### **STEP 1: Open a TCP forward port with SSH connection:**

```
root@ubuntu:~# ssh -fL 1337:localhost:1337 user2@128.4.13.229 -N
```

-L: Enables SSH local TCP port forward

*-f: Enables SSH to fork into background*

*-N: Enables SSH to run no command*

The above command will allow TCP connections on the local port 1337 of server1 to be forwarded to port 1337 of server2 through secure channel which is forked into the background with no option to run command. The -fN option is required as we require the SSH tunnel only to forward packets to the other end and not gain shell access of the other server to execute commands through that secured connection.

**STEP 2: Setting up TCP to UDP forwarding using socat on the server2:**

```
root@server2:~# socat -T10 TCP4-LISTEN:1337,fork UDP4:localhost:4789
```

The above command creates the socat on server2 to forward all packets arriving at TCP port 1337 to local UDP port 4789 at which the VXLAN service is running.

**STEP 3: Create iptables rules to redirect all VLXAN packets originating at server1 to localhost:1111 [21][22][23]:**

```
root@ubuntu:~# iptables -t nat -A OUTPUT -p udp -d 128.4.13.229 -dport 4789 -j DNAT --to-destination 127.0.0.1:1111
```

The above rule modifies all the packets having destination address 128.4.13.229:4789 to have destination address 127.0.0.1:1111, so that the packets are redirected to UDP local port 1111 from which it can be forwarded to the destination through the SSH tunnel.

The purpose of using a local port here is to have a known UDP port where the socat can listen to VXLAN packets and forward them to local SSH port. Direct forwarding of UDP packets to a TCP port is not possible without having a port-forwarding mechanism like netcat or socat.

**STEP 4: Create iptables rules at server2 to modify source address of all the packets with destination port 4789 to original source address of the VXLAN packet [22][23]:**

```
root@server2:~# iptables -t nat -A POSTROUTING -p udp -d 127.0.0.1 --dport 4789 -j SNAT --to-source 128.4.89.72:38000-45000
```

The main idea behind doing this is due to the nature of the VTEP. The VTEP sends the packet to VM only if the source IP is the IP to which it is bridged to and port be in range of 38000-45000 (from which VTEP usually selects its source port). If the source IP of packet at 4789 is “127.0.0.1:locally generated random port”, it doesn't forward it to VM and discards it as it doesn't recognize the source address 127.0.0.1

The above iptables rule makes sure that all the VXLAN packets before reaching the VTEP are modified to have the original source IP and is assigned a random port number from the given range which is typically used by OS to select a source port in a typical VXLAN communication.

**STEP 5: Setting up UDP to TCP forwarding using socat on server1 [24][30]:**

```
root@server1:~# sudo socat UDP4-LISTEN:1111,fork TCP4:localhost:1337
```

The above command sets up the socat on server1 to listen at local UDP port 1111 and forward all received packets at that port to the local TCP port 1337 from

which SSH connection is established to server2. Port 1111 is used here to receive all VXLAN packets generated at server1 which are to be transmitted to server2, and transmit them through the SSH tunnel.

### **7.3.2 For SSH tunnel 2 from server2 to server1**

#### **STEP 1: Open a TCP forward port with SSH connection:**

```
root@server2:~# ssh -fL 2323:localhost:2323 mininet@128.4.95.66 -N
```

The above command will allow TCP connections on the local port 2323 of server2 to be forwarded to port 2323 of server1 through secure channel which is forked into the background with no option to run command.

#### **STEP 2: Setting up TCP to UDP forwarding using socat on the server1:**

```
root@ubuntu:~# socat -T10 TCP4-LISTEN:2323,fork UDP4:localhost:4789
```

The above command creates the socat on server2 to forward all packets arriving at TCP port 2323 to local UDP port 4789 at which the VTEP is listening for VXLAN packets.

#### **STEP 3: Create iptables rules to redirect all VLXAN packets originating at server2 to localhost:1111**

```
root@server2:~# iptables -t nat -A OUTPUT -p udp -d 128.4.95.66 --dport 4789 -j DNAT --to-destination 127.0.0.1:1111
```

The above rule modifies all the packets having destination address 128.4.95.66:4789 to have destination address 127.0.0.1:1111, so that all VXLAN

packets are redirected to UDP local port 1111, from which it can be forwarded to the destination through the SSH tunnel.

**STEP 4: Create iptables rules at server1 to modify source address of all the packets with destination port 4789 to original source address of the VXLAN packet:**

```
root@ubuntu:~# iptables -t nat -A POSTROUTING -p udp -d 127.0.0.1 --dport 4789 -  
j SNAT --to-source 128.4.95.66:46000-56000
```

The above iptables rule modifies all the packets destined to port 4789 to have the original source IP and is assigned a random port number from the given range which is typically used by OS to select a source port in a typical VXLAN communication.

**STEP 5: Setting up UDP to TCP forwarding using socat on server2:**

```
root@server1:~# sudo socat UDP4-LISTEN:1111,fork TCP4:localhost:2323
```

The above command sets up the socat on server1 to listen at local UDP port 1111 and forward all received packets at that port to the local TCP port 2323 from which SSH connection is established to server2. Port 1111 is used here to receive all VXLAN packets generated at server2 and transmit them through the SSH tunnel.

### 7.3.3 Iptables rule

After adding above iptables rule on both the servers, they can be viewed and verified (refer figure 7.2 and figure 7.3) using the command:

```
# iptables -t nat -L
```

```

root@ubuntu:~# iptables -t nat -L
Chain PREROUTING (policy ACCEPT)
target      prot opt source                destination

Chain INPUT (policy ACCEPT)
target      prot opt source                destination

Chain OUTPUT (policy ACCEPT)
target      prot opt source                destination
DNAT        udp  --  anywhere              128.4.13.229          udp dpt:4789 to:127.0.0.1
:1111

Chain POSTROUTING (policy ACCEPT)
target      prot opt source                destination
SNAT        udp  --  anywhere              localhost              udp dpt:4789 to:128.4.13.
229:46000-56000

```

**Figure 7.2:** iptables rule at server1

```

root@server2:~# iptables -t nat -L
Chain PREROUTING (policy ACCEPT)
target      prot opt source                destination

Chain INPUT (policy ACCEPT)
target      prot opt source                destination

Chain OUTPUT (policy ACCEPT)
target      prot opt source                destination
DNAT        udp  --  anywhere              wifi-roaming-128-4-95-66.host.udel.edu
udp dpt:4789 to:127.0.0.1:1111

Chain POSTROUTING (policy ACCEPT)
target      prot opt source                destination
SNAT        udp  --  anywhere              localhost              udp dpt:4789 to:12
8.4.95.66:38000-45000

```

**Figure 7.3:** iptables rule at server2

## Chapter 8

### STUDY OF VXLAN PACKET FLOW THROUGH SSH TUNNEL

#### 8.1 Wireshark packet capture

An ICMP communication between VM 10.0.0.1 at server1 to VM 10.0.0.3 at server2 is performed successfully after configuring SSH over VXLAN tunnel. The packet capture for this communication is shown in figure 8.1 and figure 8.2 below.

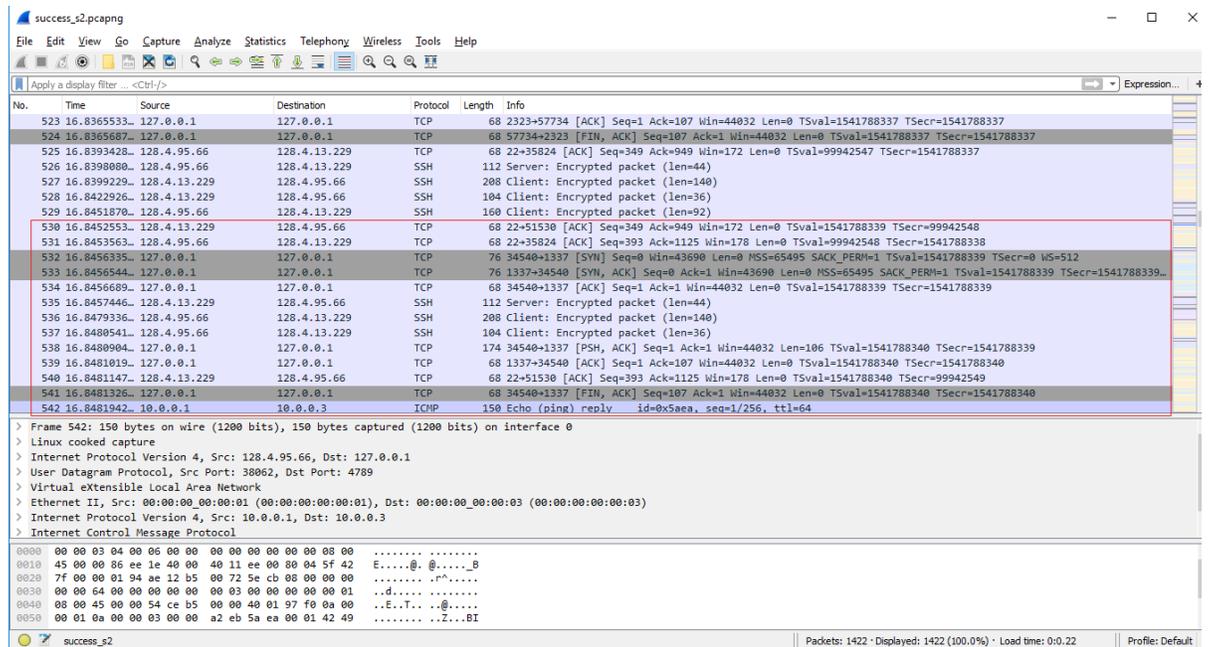


Figure 8.1: VXLAN packet capture for SSH tunnel 1 at server1

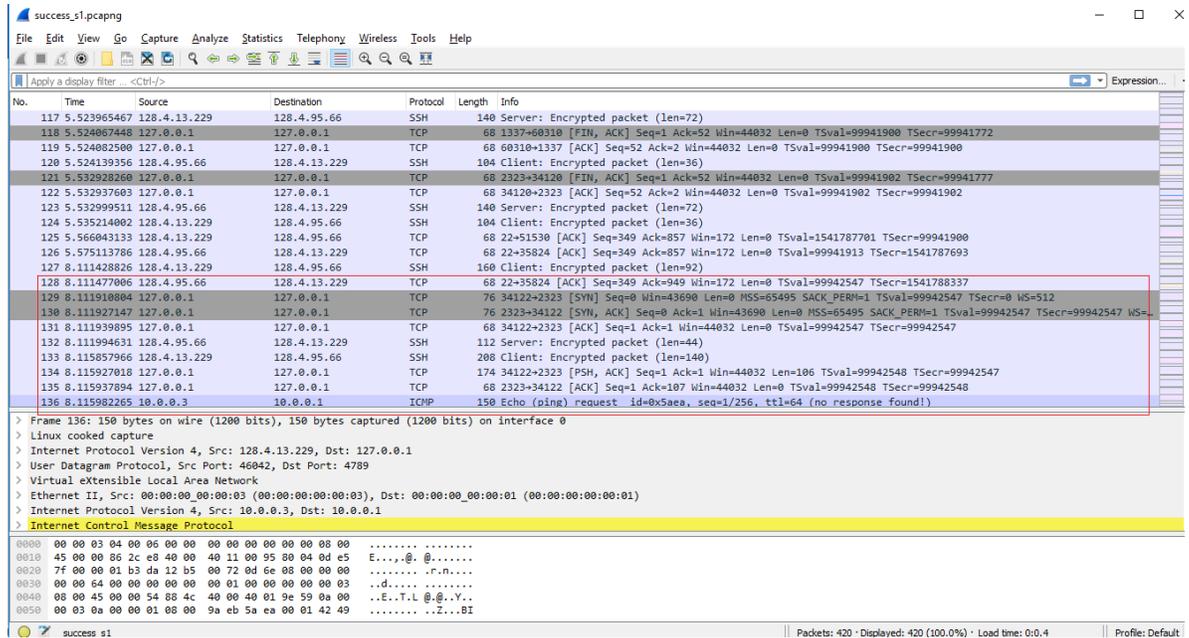
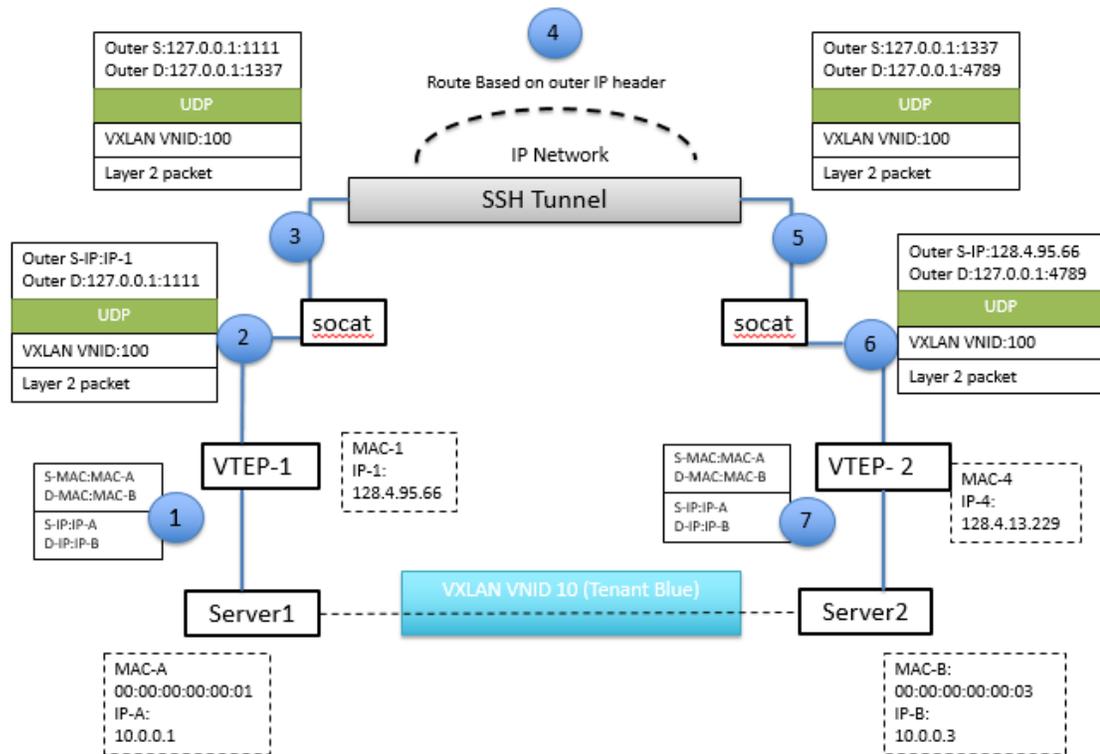


Figure 8.2: VXLAN packet capture for SSH tunnel 1 at server2

As opposed to traditional VXLAN communication where packets leaving Ethernet are unencrypted, the packets flowing through the SSH tunnel is encrypted and secured completely. Any outside intruder using a packet sniffer on this communication would not be able to see the type of packet transmitted within the SSH tunnel. Both the integrity of VXLAN UDP encapsulation header and the underlying layer 2 packet from the VM is protected at the overlay network from any potential threat.

## 8.2 Packet flow Walkthrough with SSH tunnel:

To understand the flow of packets between two VMs belonging to the same network, we consider the communication between VM h1 with IP address 10.0.0.1 and MAC address 00:00:00:00:00:01 at the server1 (128.4.95.66) and VM h3 with IP address 10.0.0.3 and MAC address 00:00:00:00:00:03 at the server2 (128.4.13.229). Note here server1 and server2 belong to different subnet and each server is connected to the other using a separate SSH tunnel. Here are the step by step process of how the packet flows from VM 10.0.0.3 to the VM 10.0.0.1 on sending a ping message from 10.0.0.3 to 10.0.0.1 as per figure 8.3:



**Figure 8.3:** Packet flow in VXLAN tunnel secured by SSH (Image adapted from [3])

- At server1, the VM h1 sends out an ICMP packet with source MAC 00:00:00:00:00:01 and destination MAC 00:00:00:00:00:03 which reaches the VTEP.
- VTEP looks at the flow table entries for a flow match. In this case it finds a match which asks packets to destination MAC address 00:00:00:00:00:03 be transmitted through output 1, which is the VTEP port that connects the VXLAN bridge to the server2. The VTEP encapsulates this packet with a UDP header with source address 128.4.95.66:46699 and destination address 128.4.13.229:4789 and VNI 100 as per the flow table.
- Since we have the iptables rule to redirect all packets destined to port 4789 to UDP local port 1111, all the VXLAN packets now reach its modified destination.
- The socat which is listening at port 1111 forwards these packets to localhost TCP port 1337 from which SSH tunnel is established to port 1337 of server2.
- The packets reaching 1337 of server2 is forwarded by the socat to UDP port 4789 at which the VTEP is listening. The packets being forwarded have source address 127.0.0.1:1337 and destination address 127.0.0.1:4789
- These packets before reaching VTEP are modified by the POSTROUTING iptables rule to have source IP address of 128.4.95.66 and a random port from the range of 35000-45000.
- The VTEP recognizes these VXLAN packets, de-encapsulates them and forwards them out of port 1 of switch s1 to VM 10.0.0.3 as per flow table entries.

- The layer 2 packet reaching 10.0.0.3 has source IP 10.0.0.1 and destination IP 10.0.0.3. The VM 10.0.0.3 then sends a ping response back to 10.0.0.1, which on reaching VTEP is UDP encapsulated with source address 128.4.13.229:58290 and destination address 128.4.95.66:4789 adding VNI number 100.
- At the server2, we have similar iptables rule as server1 to redirect packets flowing to destination port 4789 to localhost port 1111. So all the VXLAN packets are now redirected to this new address.
- The socat listening at this port then forwards it to TCP port 2323 from which server2 has established a SSH tunnel to port 2323 of server1.
- The packets reaching port 2323 of server1 would again be forwarded to localhost UDP port 4789 by the socat function.
- The packets that are being forwarded from port 1337 would have source address of 127.0.0.1:1337 and destination address of 127.0.0.1:4789
- The POSTROUTING iptables rule at server1 then modifies all the packets destined to localhost UDP port 4789, to have a source address of server2 128.4.13.229 and assign a random source port from the range 46000-56000.
- The VTEP recognizes these VXLAN packets, de-encapsulates them and forwards them out of port 1 of switch s1 to VM 10.0.0.1 as per flow table entries. The layer 2 packets from VM 10.0.0.3 now reaches back VM 10.0.0.1 successfully.

## **Chapter 9**

### **DISCUSSION**

VXLAN provides mechanisms used on overlay networks to aggregate and tunnel multiple layer 2 networks across a layer 3 infrastructure, to make them look like a part of the same layer 2 network even though they are physically placed on different servers on different subnet. VXLAN technology is mainly used to address the scalability issues associated with large cloud computing environments, where it uses 24-bit segment ID known as Virtual Network Identifier (VNI) which enables upto 16 million VXLAN segments to be a part of the same administrative domain.

The security measures of VXLAN communication over layer 3 network has not been addressed in VXLAN documentation and requires outside traditional security mechanisms to authenticate and optionally encrypt VXLAN packets. Implementation of a simpler and commonly used security mechanism like SSH encryption to protect VXLAN data integrity has not been tried till date. The goal of the study is to find a way to incorporate the SSH tunneling feature over the VXLAN tunnel to secure all communication through that tunnel.

The primary step of the study is to configure a VXLAN tunnel between two VMs sitting on different datacenters but belonging to same domain. We take two Ubuntu servers for this purpose which would host VMs that are configured to be a part of the same network. The first approach was to follow the guidelines for VXLAN installation on Ubuntu machine provided by Open vSwitch installation resources

[26][27], which provides steps to install Open vSwitch on both the servers and manually setup OVS bridge s1 and add the Ethernet port to the bridge so that all the traffic out of the server is passed through the bridge. The bridge s1 is also given two additional vports to which the two VMs would be connected to. KVM is used to spin up the VMs whose network interfaces are added to the vports. Next, the flow table entries to direct the packet flow to the correct output port is added to each of the bridge on both sides. The two Ubuntu servers have reachability to each other over public internet which was verified by sending ICMP messages to each other. The VTEP on each server knows how to forward the packets depending on the layer 2 destination MAC address. Ideally the layer 2 packet from the VM in the server after encapsulation by the VTEP should be transmitted over layer 3 network to the other server where its VTEP is listening at port 4789 for VXLAN packets. But on practical implementation in this study, this does not occur. The VXLAN packets do not reach port 4789 of the other server. The reason behind this behavior could not be justified strongly due to lack of resources on the issue. It can only be speculated to be due to a missed VTEP configuration at the server ends, because the VMs belonging to the same network domain and residing on the same server could communicate with each other. So the OVS bridge does effectively route traffic based on the flow entries to the correct output port. But in case of the VM which belong to the same network domain and reside in different server, the VTEP doesn't receive the VXLAN packets.

Mininet on the other hand provided a quick and easy means to simulate a network topology consisting of a controller and a OVS switch with hosts attached to its port. Mininet was developed to aid laboratory research on SDN and to understand and explore OpenFlow better. It uses Open vSwitch in its topology and all the network

configuration that we require in our study is provided on mininet. Hence, mininet was used to create OVS switch along with VM hosts on each of the server to implement the VXLAN tunnel.

The use of mininet on Ubuntu servers having public address, for implementation of VXLAN tunnel, was successful as the two VMs on either server belonging to the same network could communicate with each other. We could see VXLAN packets being transmitted and received at the VTEP using the Wireshark packet capture.

The important section of this study comes at this stage where we try to incorporate an encryption mechanism to protect the integrity of the VXLAN packets over public internet. Options like IPsec and similar encryption mechanism require additional hardware resources like router and switch which needs to be configured to implement encryption and authentication mechanisms to the transmitting packets. The idea is to look for an encryption option that can be implemented right at the server rather than configuring routers. SSH tunneling is one such mechanism that fits this requirement and it's easy to implement on the host machines and requires no additional network configuration. It is easy to implement at industry level too than just laboratory environments. Hence, we tried SSH mechanism on our VXLAN tunnel to see if it can successfully transport VXLAN packets through secured network to the destination.

SSH local port forwarding is used to create a secure connection between a local TCP port at server1 and at server2. The validity of this secure connection was tested on a TCP service since SSH works on TCP port by default. So, a http web server is launched at server2, which listens at port 8080 using the command:

```
echo "Hello, World" > index.html
```

```
nohup busybox httpd -f -p 8080 &
```

A SSH local forward connection is made from server1 TCP port 1234 to server2 port 8080. Now on opening the browser on server1 to `http://127.0.0.1:1234` displays the web page *"Hello World"*. This proves that SSH local port forwarding works for TCP. Now this same SSH tunnel is used for UDP application where a txt file is transmitted to the other server using socat. After creating the SSH tunnel between the servers, socat is opened on UDP port 5656 which forwards packets to SSH port 1337 which inturn tunnels to port 1337 of the other server. The socat on the other server forwards the packet received at 1337 to its local UDP port 8989. The txt file on forwarding to UDP port 5656 is received at server2 UDP port 8989 successfully. This proves the usefulness of SSH tunnel for UDP packet forwarding. Since VXLAN uses UDP encapsulation, this technique should work for VXLAN tunneling too.

Setting up SSH tunnel over VXLAN tunnel required two separate SSH port forwarding, one from server1 to server2 and the other from server2 to server1 as described in the chapter 6. Socat is used here to facilitate the UDP to TCP port forwarding to the SSH port. We use iptables rule to redirect all VXLAN packets with destination address of the other server to reach localhost port at which socat is listening. One method of doing it was to assign different destination ports to the VTEPs on both the servers, like server1 VTEP be listening to VXLAN packets on port 7654 and server2 VTEP be listening to VXLAN packets on port 4789. By doing this, we can set iptables rule to individually access packets destined to go to the other VTEP port 7654 to be redirected to localhost:1111, also setting similar rule on server2

where packets destined for port 7654 be redirected to localhost:1111 where socat is listening and not interrupt the packets coming to 4789. This method traditionally works on any other service like HTTP or SMTP since they are a client server application, where the service is up at the server and the service request is done from client side. Changing the destination port is effective as the listening port at the server side is altered and now, the request must be made to the new port from client side. But this doesn't work for VXLAN application since VXLAN service is running on both the hosts involved in the tunnel where the VTEP listens to VXLAN packets on both ends. We cannot set different VXLAN destination port for VTEP on each side since VXLAN communicates on one destination port on both side of the tunnel. It has the default port of 4789 but it can be changed using the command option:

```
$ ovs-vsctl add-port br0 vxlan1 -- set interface vxlan1 type=vxlan \  
options:remote_ip=192.168.1.2 options:key=flow options:dst_port=8472
```

Changing this port cause the VTEP at one server to listen to VXLAN packets at port 8472 and transmit packets to port 8472 of the other server through the tunnel. If both tunnel end VTEPs are configured to have different destination ports, each one would transmit packets to different destination ports at which the destination server is not listening to for VXLAN packets. Ultimately those packets are dropped and communication fails. Hence, it is mandatory that the VTEPs on both end of the VXLAN tunnel be listening at the same port.

The other idea was to keep the destination port of VXLAN unmodified and redirect all packets having destination IP of server2 to localhost:1111 where socat is listening. This iptables rule works well and all the VXLAN packets destined to the other server now reaches socat at port 1111 which forwards them to SSH port 1337

which inturn securely tunnels to port 1337 of server2. The socat at server2 forwards the incoming packets to UDP port 4789 to reach the VTEP. This internal forwarding of packets keeps modifying the source and destination address of the packet at each transmission. The result of this has a VXLAN packet reaching the destination VTEP with source address of 127.0.1:1337 and destination address 127.0.0.1:4789. Though technically the packet reaches port 4789, the VTEP fails to recognize these packets as they no more preserve the original source address and it could be a packet from any other source which is not a part of the VXLAN tunnel. Hence, the VTEP drops this packet in the name of damaged VXLAN packet and doesn't de-encapsulate them further and nor does they send a response packet to the other server.

VTEP on receiving VXLAN packets verifies both the source IP to match the address to which it is tunneled to via Ethernet output. The OS selects a random source port for VXLAN from a range around 35000 to 55000. We verify this by running VXLAN communication over different periods of time and observing the source port number for the packets. For every new tunnel setup, the OS chooses a separate source port for its VXLAN packets. This behavior of VTEP for verifying source address was also tested by first altering just the destination port of the VXLAN packets reaching the destination VTEP using the following command [22].

```
root@ubuntu:~# iptables -t nat -A POSTROUTING -p udp -d 127.0.0.1 --dport 4789 -  
j SNAT --to-source 127.0.0.1:46000
```

This means that all packets with destination 127.0.0.1:4789 and source 127.0.0.1:1337 would be changed to have source address 127.0.0.1:46000 (a random port number from the range used for VXLAN). The VTEP fails to identify the packets and de-encapsulation fails to occur in this case. On another scenario, only the source

IP of the VXLAN packet reaching VTEP was changed to have the original server IP without altering the source port as shown below.

```
root@ubuntu:~# iptables -t nat -A POSTROUTING -p udp -d 127.0.0.1 --dport 4789 -  
j SNAT --to-source 128.4.95.66
```

So the packets reaching the VTEP of server2 would have source IP of server1 (128.4.95.66) and port 1337 (since the packet was forwarded by socat from the SSH port). Even in this case, the VTEP fails to recognize the packet. Then on the next scenario, on keeping the original source IP and port of the packet for VTEP intact as in the command shown below, it is observed that the packet de-encapsulation occurs and a layer 2 response is sent back from inner VM which is encapsulated by the VTEP and transmitted back again.

```
root@ubuntu:~# iptables -t nat -A POSTROUTING -p udp -d 127.0.0.1 --dport 4789 -  
j SNAT --to-source 128.4.95.66:46000
```

Now as this technique works perfectly, we make iptables rule on both server to make the necessary alterations to the VXLAN packets reaching VTEP for it to recognize as a legit packet and send a response back. We use a simple ICMP communication on this tunnel setup to test its efficiency. On sending a ping from VM 10.0.0.1 on server1 to VM 10.0.0.3 on server2, a destination unreachable error is still found. On analyzing the packet capture, it is seen that using a random source port works with ARP broadcast that the VM sends prior to ICMP message to receive MAC address of VM 10.0.0.3. The VM 10.0.0.3 responds with a ARP reply, but the ICMP message send by 10.0.0.1 after receiving ARP reply is dropped by VTEP at server2. The reason behind this behavior is found to be the usage of same source port for the VXLAN packet for both ARP and ICMP messages. It is later learnt that during

VXLAN tunnel communication, the VTEP at source end uses a random OS generated port number for each of the protocol messages. Which means that on sending an ICMP message on a traditional VXLAN tunnel without SSH, the source port for VXLAN packet having ARP request is different from source port for VXLAN packet having the ICMP message. The solution to the error found in the previous case is fixed by assigning different random source port numbers for the VXLAN packets reaching VTEP. This would differentiate the packets containing ARP exchange and the ICMP exchange. This is achieved by using the command option [22]:

```
root@ubuntu:~# iptables -t nat -A POSTROUTING -p udp -d 127.0.0.1 --dport 4789 -  
j SNAT --to-source 128.4.95.66:46000-56000
```

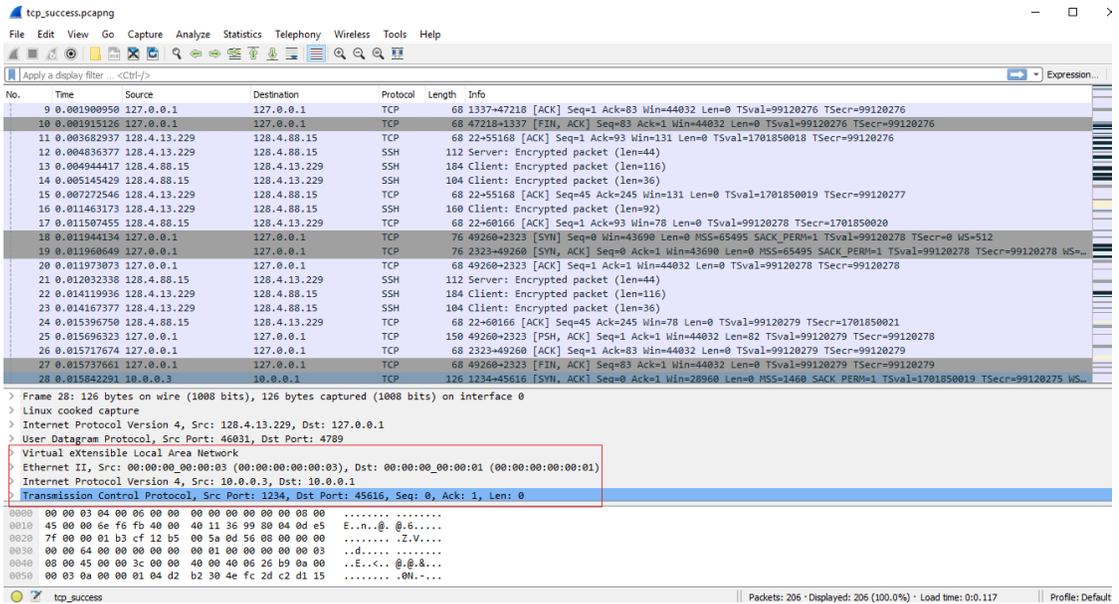
On applying this rule on both sides on the tunnel, a successful ICMP reply for VM 10.0.0.1 from 10.0.0.3 is seen. This proves our study on VTEP behavior true and to prove the usability of the proposed configuration with other protocols, we test it next with a TCP communication between the two VMs. For this, a HTTP server at 10.0.0.3 with a test index page “Hello there” is created using the command:

```
echo "Hello there" > index.html  
nohup busybox httpd -f -p 1234 &
```

A http get request is made from VM 10.0.0.1 to 10.0.0.3 to display the web page content on the shell using the command [31]:

```
wget -O - - http://10.0.0.3:1234
```

A successful display of index page content is achieved on the shell and the packet capture shows the transmission of VXLAN packets through the SSH tunnel as seen in figure 9.1 below.



**Figure 9.1: TCP through SSH tunnel**

This proves the validity of the SSH tunnel configuration for most of the communication protocols. Irrespective of the type of layer 2 packet, VTEP transmits it through the SSH tunnel after UDP encapsulation.

The security technique discussed in this study requires usage of additional UDP port to run socat which may make these ports vulnerable to possible open port exploitations, but this is not considered a major security threat and usage of proper firewall rules and packet filters would help eliminate this danger.

## **Chapter 10**

### **CONCLUSION**

VLAN is becoming the new interest in the datacenter these days owing to its focus on the network infrastructure and its ability to allow 16 million logical segments which is needed to cope up with the increase in demand for server virtualization.

VLAN can be implemented in the network using hardware switches like Nexus 9000 as used by Cisco or can be virtually implemented using VMware NSX and Open vSwitch. Securing the VLAN communication over layer 3 is one major concern as VLAN doesn't provide data protectivity. There are several security mechanisms to uphold data integrity. The purpose of the study is to explore the usability of one such simple security mechanism SSH over VLAN tunnel and test its effectiveness in encrypting messages so that the need for extensive firewall devices can be saved.

The new proposed network configuration uses Open vSwitch for OpenFlow implementation and a pair of VM each belonging to servers that are on different network. The SSH tunnel is configured without altering the VLAN tunnel configuration and the tunnel is tested by sending layer 2 packets between the VMs. The VLAN packets transmitted through this network is secured by the SSH tunnel and the integrity of the packet is preserved. The operation of the VLAN layer 2 over layer 3 is also maintained effectively. This successful study now provides one new and easy way for secure VLAN communication.

## **Chapter 11**

### **FUTURE DIRECTIONS**

The topology used in this study is restricted only to a one to one communication network between two VMs using a VXLAN tunnel. So, manual configuration of a SSH tunnel over the VXLAN tunnel doesn't seem to be a tedious task. However, in cases of a more complex topology where many VXLAN tunnels are configured between multiple servers in different datacenters, creating multiple SSH tunnels between the physical servers would be tedious and difficult to manage. Hence, to help the practical implementation of this security mechanism at industry level, a user interface application that could automate the creation of SSH tunnel in the background along with addition of appropriate iptables rule to redirect those VXLAN packets through that SSH tunnel, upon giving the source and destination address for the VXLAN tunnel as input to the interface would be of great use. As there are typical user interface application like Putty, development of an application to set up a SSH tunnel between VTEPs of the source and destination machines would ease the work of configuring every network setup manually. It will also provide a faster way to create and manage secure VXLAN communications between the servers, without having to consider human errors during manual multiple network configuration.

Creation of UDP to SSH TCP port forwarding, redirection of VXLAN packets through SSH tunnel and modification of source address of packets reaching destination VTEP to the original address are all essential aspects that must be taken into consideration while building this tool. The tool must be able to let the user view

and control the various SSH tunnel encapsulating each VXLAN tunnel and terminate the tunnel whenever they need. This could make the use of the tunnel securing process more convenient to handle and manage the network better.

## BIBLIOGRAPHY

1. Growth of Cloud demand <http://fortune.com/2017/02/22/cloud-growth-forecast-gartner/>
2. VXLAN IETF draft <https://tools.ietf.org/html/rfc7348>
3. VXLAN CISCO overview  
<http://www.cisco.com/c/en/us/products/collateral/switches/nexus-9000-series-switches/white-paper-c11-729383.html>
4. Arista VXLAN white paper  
[https://www.arista.com/assets/data/pdf/Whitepapers/Arista\\_Networks\\_VXLAN\\_White\\_Paper.pdf](https://www.arista.com/assets/data/pdf/Whitepapers/Arista_Networks_VXLAN_White_Paper.pdf)
5. Introduction to VXLAN <http://www.therandomsecurityguy.com/vxlan/>
6. VXLAN: A framework for overlaying virtualized layer 2 networks over layer 3 networks <http://tools.ietf.org/html/draft-mahalingam-dutt-dcops-vxlan-07>
7. Software Defined Networking  
<https://www.sdxcentral.com/sdn/definitions/what-the-definition-of-software-defined-networking-sdn/>
8. OpenFlow overview <https://www.sdxcentral.com/sdn/definitions/what-is-openflow/>
9. Openflow Switch specification  
<https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.1.pdf>
10. OpenFlow white paper <http://archive.openflow.org/documents/openflow-wp-latest.pdf>
11. Open vSwitch overview <https://www.sdxcentral.com/cloud/open-source/definitions/what-is-open-vswitch/>
12. Open vSwitch documentation <http://docs.openvswitch.org/en/latest/>

13. Mininet overview <http://mininet.org/overview/>
14. Lantz, B., Heller, B., & McKeown, N. (2010, October). A network in a laptop: rapid prototyping for software-defined networks. In Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks (p. 19). ACM.
15. Mininet sample workflow <http://mininet.org/sample-workflow/>
16. VXLAN does not define Security on overlay  
<https://www.packetmischief.ca/2013/12/03/five-functional-facts-about-vxlan/>
17. UDP in SSH tunneling <http://zarb.org/~gc/html/udp-in-ssh-tunneling.html>
18. VXLAN Tunnel End Points  
[http://www.cisco.com/c/en/us/td/docs/switches/datacenter/nexus3000/sw/interfaces/6\\_x/b\\_Cisco\\_n3k\\_Interfaces\\_Configuration\\_Guide\\_602\\_U11/b\\_Cisco\\_n3k\\_Interfaces\\_Configuration\\_Guide\\_602\\_U11\\_chapter\\_01000.html](http://www.cisco.com/c/en/us/td/docs/switches/datacenter/nexus3000/sw/interfaces/6_x/b_Cisco_n3k_Interfaces_Configuration_Guide_602_U11/b_Cisco_n3k_Interfaces_Configuration_Guide_602_U11_chapter_01000.html)
19. Mininet topology configuration <http://mininet.org/walkthrough/>
20. Setting up VXLAN tunnel <http://networkstatic.net/setting-overlays-openvswitch/>
21. Local port forwarding using iptables  
<https://stackoverflow.com/questions/28170004/how-to-do-local-port-forwarding-with-iptables>
22. Framing iptable rules  
<http://www.netfilter.org/documentation/HOWTO/NAT-HOWTO-6.html>
23. Viewing Iptable rules  
<https://www.digitalocean.com/community/tutorials/how-to-list-and-delete-iptables-firewall-rules>
24. Socat implementation  
<https://superuser.com/questions/53103/udp-traffic-through-ssh-tunnel>
25. Reyes, G. P., Dammers, M., & Kastanja, M. (2014). *Security assessment on a VXLAN-based network* (Doctoral dissertation, Master's thesis, University of Amsterdam, Amsterdam).
26. VXLAN setup  
<http://blog.arunsriraman.com/2017/02/how-to-setting-up-gre-or-vxlan-tunnel.html>

27. Open vSwitch cheat sheet  
<http://therandomsecurityguy.com/openvswitch-cheat-sheet/>
28. Nakagawa, Y., Hyoudou, K., & Shimizu, T. (2012, August). A management method of IP multicast in overlay networks using openflow. In *Proceedings of the first workshop on Hot topics in software defined networks* (pp. 91-96). ACM.
29. Ketikci, F., & Askar, S. (2015, February). Emulation of Software Defined Networks Using Mininet in Different Simulation Environments. In *Intelligent Systems, Modelling and Simulation (ISMS), 2015 6th International Conference on* (pp. 205-210). IEEE.
30. Socat overview  
<http://www.dest-unreach.org/socat/doc/socat.html>
31. Display web page content on shell  
<https://www.cyberciti.biz/faq/unix-linux-get-the-contents-of-a-webpage-in-a-terminal/>