European Commission | Horizon 2020 European Union funding for Research & Innovation

Cyber Security PPP: Addressing Advanced Cyber Security Threats and Threat Actors

Cyber Security Threats and Threat Actors Training - Assurance Driven Multi- Layer, end-to-end Simulation and Training

D2.3: Interlinking of Emulated Components Modules v1 †

Abstract: This deliverable provides a technical description of the design and the development of the functionalities that support the generation of the virtual network infrastructure connecting the emulated components generated by the Emulation Tool. This Deliverable is the result of the first iteration of task T2.3 activities.

Contractual Date of Delivery	31/08/2019
Actual Date of Delivery	31/08/2019
Deliverable Security Class	Public
Editor	*Ernesto Damiani, Elvinia Riccobene, Stelvio Cimato, Chiara Braghin, Claudio Ardagna, Fulvio Frati, Sadegh Astaneh, Lara Mauri (UMIL)*
Contributors	*UMIL, FORTH, ITML*
Quality Assurance	*Oleg Blinder (IBM), Vassilis Prevelakis (TUBS)*

† *The research leading to these results has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 786890.*

The *THREAT-ARREST* Consortium

Foundation for Research and Technology – Hellas (FORTH)	Greece
SIMPLAN AG (SIMPLAN)	Germany
Sphynx Technology Solutions (STS)	Switzerland
Universita Degli Studi di Milano (UMIL)	Italy
ATOS Spain S.A. (ATOS)	Spain
IBM Israel – Science and Technology LTD (IBM)	Israel
Social Engineering Academy GMBH (SEA)	Germany
Information Technology for Market Leadership (ITML)	Greece
Bird & Bird LLP (B&B)	United Kingdom
Technische Universitaet Braunschweig (TUBS)	Germany
CZ.NIC, ZSPO (CZNIC)	Czech Republic
DANAOS Shipping Company LTD (DANAOS)	Cyprus
TUV HELLAS TUV NORD (TUV)	Greece
LIGHTSOURCE LAB LTD (LSE)	Ireland
Agenzia Regionale Strategica per la Salute ed il Sociale (ARESS)	Italy

Document Revisions & Quality Assurance

Internal Reviewers

1. *Oleg Blinder (IBM),*
2. *Vassilis Prevelakis (TUBS)*

Revisions

Version	Date	By	Overview
1.0	30/08/2019	UMIL	Final version
0.7	28/08/2019	UMIL	Deliverable updated after internal reviews
0.5	31/07/2019	UMIL	Deliverable ready for internal review
0.3	15/07/2019	UMIL	Draft contribution
0.2	21/06/2019	FORTH	FORTH's initial contribution
0.1	01/05/2019	Editor	First Draft

Executive Summary

This deliverable provides a technical description of the design and the development of the functionalities that support the generation of the virtual network infrastructure connecting the emulated components generated by the Emulation Tool.

The activities of the first year have been dedicated to the definition of the mechanism to *(i)* connect the emulated components defined in task T2.1 using a simple network and provide access to them, and *(ii)* define the algorithms and rules to deploy complex network infrastructures.

The work of this task has been strictly correlated with the activities of task T2.1.

Table of Contents

List of Abbreviations

CIDR Classless Inter-Domain Routing

CTTP Cyber Threat and Training Preparation

DMZ Demilitarized Zone

GRE Generic Routing Encapsulation

HOT HEAT Orchestration Template

IP Internet Protocol

OS Operating System

RDP Remote Desktop Protocol

SSH Secure Shell

TCP Transmission Control Protocol

UUID Universally Unique Identifier

VLAN Virtual Local Area Network

VM Virtual Machine

VNC Virtual Network Computing

VXLAN Virtual Extensible LAN

WP Work Package

XML eXtensible Markup Language

XSD XML Schema Definition

List of Figures

1 Introduction

The interlinking of the emulated components plays a role of paramount importance in cyber range training environments. In fact, trainees have to face real-world cybersecurity threats that will be applied in virtual environments mimicking the real working environment (e.g. (Ferrera et al., 2018; Hatzivasilis et al., 2017; Hatzivasilis et al., 2019; Cesena et al., 2017). This gives them the possibility to improve their knowledge and to apply countermeasures to real attacks.

The Emulation Tool will embrace in full the model-driven approach of THREAT-ARREST. The definition of the whole emulated environment, as well as the Virtual Machines (VMs) interlinking system, is done by the compiling and execution of the Cyber Threat and Training Preparation (CTTP) Emulation sub-model, i.e., a specific view of the CTTP model that takes into consideration only the objects playing a role in the emulation infrastructure.

The objective of this deliverable is to describe the network aspect of the emulated environment that has been defined in "D2.1 – Emulated components Generated Module v1", giving an overview of the technologies used in the development and of the algorithm that will drive the final development of the Emulation Tool. In particular, this deliverable provides a technical description of the design and the development of the functionalities that support the generation of the virtual network infrastructure connecting the emulated components generated by the Emulation Tool.

It is the result of the first iteration of task T2.3 activities, where we defined the rules for the deployment of virtual networks inside the Emulation Tool, characterized by a complex topology reflecting the topology of Pilots' network, paving the way for the second iterations and the updated version of the tool.

The deliverable covers also the interactions between the trainee (or the trainer) and the VM. A specific tool, Apache Guacamole, has been introduced in the infrastructure and works as transparent gateway providing them with a Secure Shell (SSH) or Remote Desktop connections through a web page.

The deliverable is organized as follows. First, in Section 2 we give to the reader an overview of the final use case, in order to introduce the context and the problem we had to face in order to supply the emulated components with an actual network environment.

Then, in Section 3 we describes the frameworks needed to deploy it. We start from the requirements and platform reference architecture analysis delivered by tasks T1.2 and T1.3 and described in D1.2 and D1.3 released, respectively, at M4 and M6 of the project. Here, OpenStack has been chosen has reference architecture for the emulated environment, basing on it our activities.

Finally, Section 4 describe the algorithms that will drive the development of the Emulation Tool in the second iteration of task T2.3, for what that directly concerns the deploying of the virtual network infrastructure. Section 5 draws our conclusions. Appendix I details a HEAT template that instantiates a smart vessel scenario that is detailed in the previous sections.

2 CTTP-Model Driven Emulation

In this section, we present the application of the CTTP models, which will be described in the deliverable D3.1, to the Emulation Tool. Please note that the complete syntax and structure of the CTTP models will be considered for the implementation of version 2 of the Emulation Tool.

In the following, we present a use case that is directly connected with the use case presented in Section 4 of D2.1. In fact, in the deliverable D2.1, we presented the process of instantiating emulated components based on the CTTP model.

In brief, the model describes the hardware, software, and infrastructure primitives that are required in order to implement the desired functionality for an emulated component. Installation scripts are also included in the model, detailing how these components can be developed in OpenStack.

This section further describes how to define the networking infrastructure of a modelled pilot system. The modelling of the internal interaction between the emulated components and the rest of THREAT-ARREST platform modules is also determined here, with the main interconnection functionality being presented in D2.4.

2.1 Networking Capabilities

In OpenStack, we can model two main connection types: *i)* internal within the platform, and *ii)* external communication. In the following example, we instantiate the emulated components of a vessel for the Smart Shipping scenario (Use Case 3).

In the scenario, two VMs will be deployed. The first one represents the Captain's PC (Windows 10 OS, 20GB hard-disk, 16GB RAM, Internet Explorer, Windows Defender anti-virus). The modelling and deployment for this VM is detailed in the deliverable D2.1. The second instance is the on-deck equipment (Ubuntu 19 OS, 20GB hard-disk, 16GB RAM, Jasima simulator). This is a VM that runs the Jasima simulator (SimPlan AG, 2019) which simulates the ship's navigation modules and their on-deck monitors. The CTTP modelling of the simulation is detailed in the deliverable D5.1 while the VM can be instantiated in a similar fashion as the captain's PC (deliverable D2.1).

In this example, we consider that the OpenStack modelling requirements for two VMs have been completed on the basis of the description provided in the aforementioned deliverables (e.g. images, flavors, network interfaces, etc.), and now, the goal is to deploy a network with them.

The two VMs are connected in a local network, which is formed as an internal connection in OpenStack. Then, the Captain's PC can communicate to Internet through a satellite link, which is defined as an external OpenStack connection. Next figure depicts the deployed infrastructure and the interconnection of the various components.

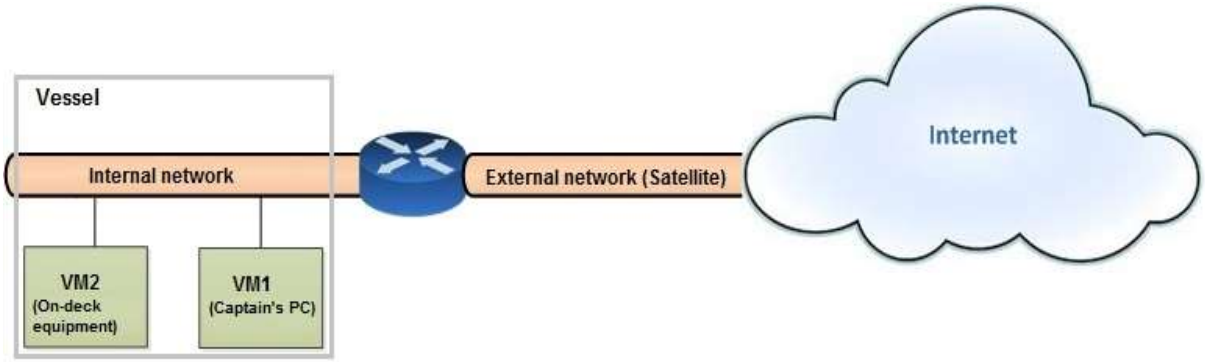

Figure 1 Smart vessel emulation in OpenStack

An example of template that can implement this infrastructure is detailed in Appendix I. This template deploys the whole network in once and it can be included in the instantiation script for an emulated vessel asset as described in the deliverable D2.1.

Alternatively, the two VMs can be instantiated individually, based on D2.1 and D5.2, but the same internal (private) network configurations must be included in order to deploy their local connection.

3 Emulation Tool Infrastructure

OpenStack has been chosen as the foundation framework of the Emulation Tool for its widespread diffusion and adoption, and for its modularity, which allows it to be applied for multiple specific scenarios. Requirement analysis completed within the activities of the work package (WP) 1 has selected it as the best candidate for the emulated environment (Figure 2).

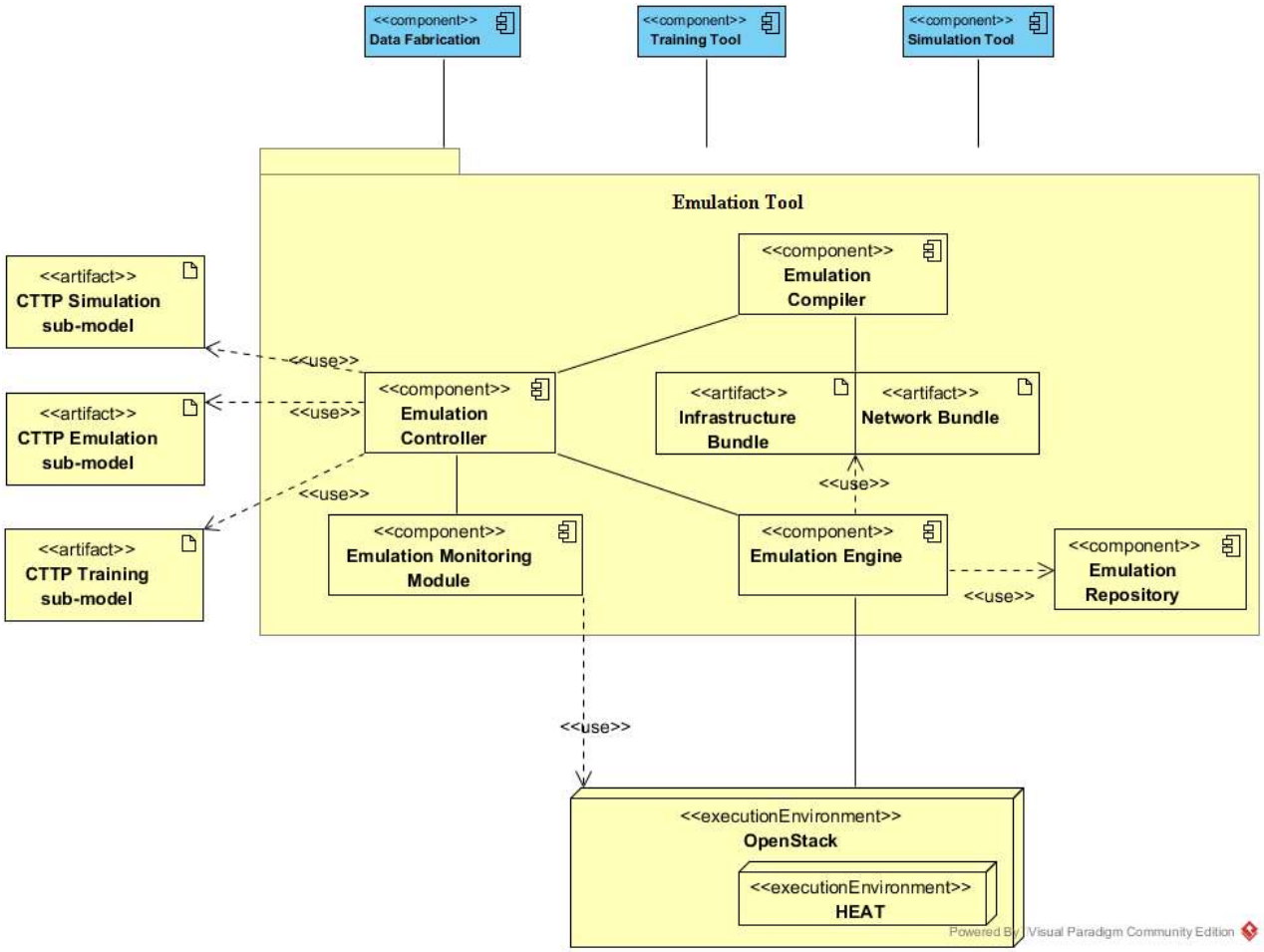

Figure 2: Emulation Tool Infrastructure.

With respect to the virtual network management, two projects that are part of the OpenStack framework will be mainly involved:

- **HEAT**, the OpenStack Orchestrator module, and
- **Neutron**, the virtual network management module.

In the following sections, a description of Neutron and HEAT, with respect to their role in the Emulation Tool, are provided. It is important to note that the final objective of task T2.3 activities is to give to trainees a real-world virtual environment with the same network infrastructure they can find in their actual working environment.

3.1 Neutron

Neutron is an OpenStack project that provides specific APIs to supply "network connectivity as a service" and that is in charge of the virtual network interfaces managed by other OpenStack services, like for instance Nova VMs (OpenStack, 2019), allowing them to create, attach and use virtual device interfaces to the networks.

Furthermore, Neutron allows the creation of complex virtual network topologies supplying specifying objects abstractions to manage subnets, routers, firewall, load balancers, and virtual private networks. Each abstraction has a functionality that mimics its physical counterpart: networks contain subnets, and routers route traffic between different subnets and networks.

Neutron also supports Security Groups. A Security Group defines the set of network operations the associated object can apply to the network. A VM can belong to one or more groups, and the network layer applies the rules in those security groups to block or unblock ports, port ranges, or traffic types for that VM.

Neutron has strict interactions with other OpenStack modules to manage specific aspects of networking, namely:

- OpenStack Identity service (*Keystone*), used for authentication and authorization of API requests;
- OpenStack Compute service (*Nova*), used to plug each virtual network interface of the VM into a network.
- OpenStack Dashboard (*Horizon*), used to create and manage network services through the web-based graphical interface.

3.2 Neutron Networking with HEAT

OpenStack provides resource types specific of the Neutron module that can be exploited in HOT templates to define custom networks, and plug virtual interfaces into them. Figure 3 depicts the structure of a minimal Neutron virtual network.

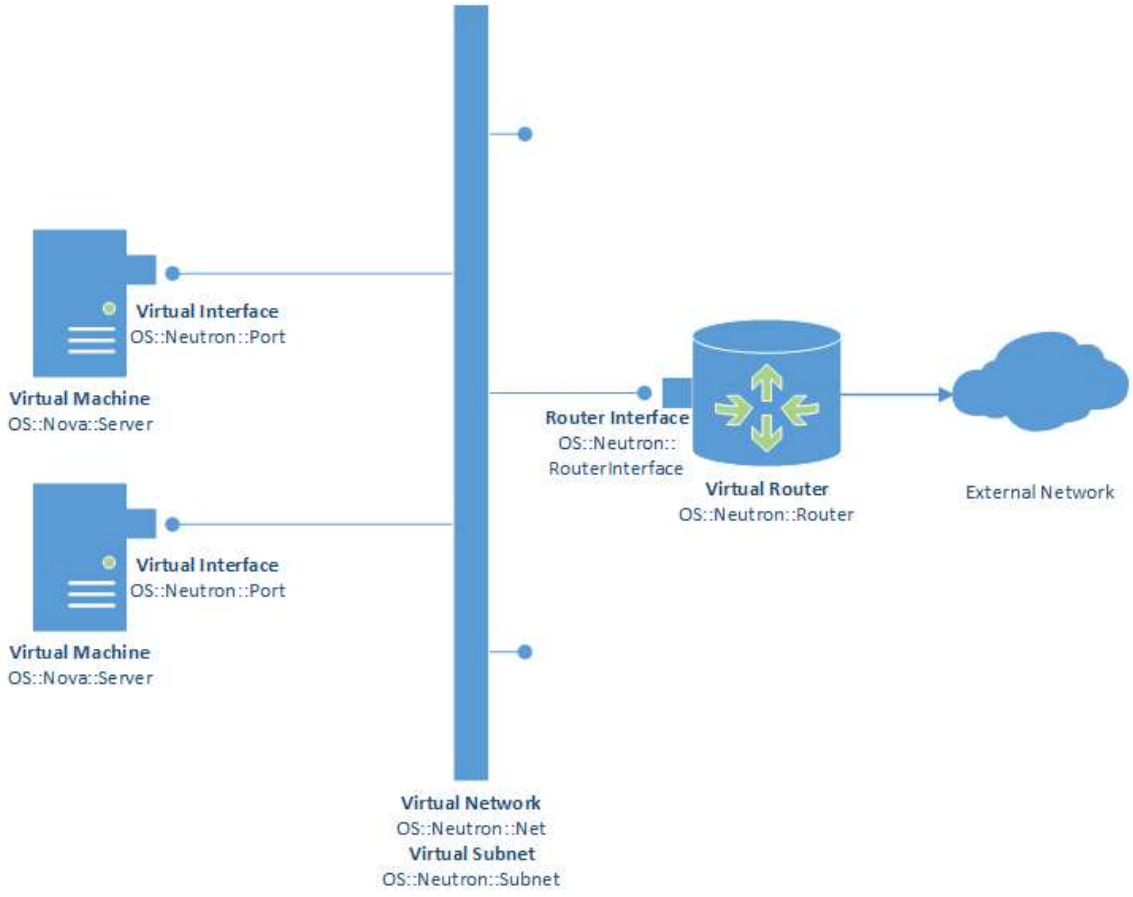

Figure 3: Neutron minimal network.

Each VM is associated to one or more objects of type *OS::Neutron::Port* that virtualize network interfaces functionalities. The ports will be interpreted by the VM as common network interfaces, i.e. *ethX* in Linux.

Ports are connected to instances of type *OS::Neutron::Net* that provide connectivity within projects. By default, they are fully isolated and are not shared with other projects. The following types of network isolation and overlay technologies are supported by OpenStack and defined at creation-time:

- **Flat:** all instances reside on the same network, which can also be shared with the hosts. No Virtual Local Area Network (VLAN) tagging or other network segregation can take place.
- **VLAN:** allows users to create multiple provider or project networks using VLAN IDs (802.1Q tagged). Instances can communicate with each other across the environment with dedicated servers, firewalls, load balancers, and other networking infrastructure on the same layer 2 VLAN.
- **Generic Routing Encapsulation (GRE) and Virtual Extensible LAN (VXLAN)**: encapsulation protocols that create overlay networks to activate and control communication between compute instances. A Networking router is required to allow traffic to flow outside of the GRE or VXLAN project network. A router is also required to connect directly-connected project networks with external networks, including the Internet.

Networks can be easily partitioned using the type *OS::Neutron::Subnet* that enables the creation of subnets, with a specific address pool, within a network.

Routers are then required to provide connectivity to the Internet through the External network. The type *OS::Neutron::Router* allows instance creation. Each router is associated to an external network and can be connected to one or more network using objects of type *OS::Neutron::RouterInterface*.

The schema in Figure 3 is deployed trough HEAT using the template in Figure 4. Specific information about the resource types and input parameters are provided in the deliverable D2.1. The parameter *public_net1* refers to the external network, already defined in OpenStack, which gives access to Internet and is described by its Universally Unique Identifier (UUID).

```
description: 'Stack of scenario Threatarrest '
heat_template_version: '2018-08-31'
parameters:
  flavor-2048-1-20-cirros-0.4.0-x86_64-disk:
    default: flavor-2048-1-20-cirros-0.4.0-x86_64-disk
    type: string
  gateway_internal:
    default: 20.20.0.254
    type: string
  cidr1:
    default: 20.20.0.0/24
    type: string
  img1:
    default: cirros-0.4.0-x86_64-disk
    type: string
  key1:
    default: common_key
    type: string
  public_net1:
    default: 0ec0b875-0a61-460e-be46-8214cd9a6d5c
    type: string
  secgroup1:
    default: f9a3fe5a-1d5b-40e6-b627-f0a9a1f57ce6
    type: string
resources:
  vm1:
    properties:
      flavor:
        get_param: flavor-2048-1-20-cirros-0.4.0-x86_64-disk
      image:
        get_param: img1
      key_name:
        get_param: key1
      name: vm1
      networks:
      - port:
          get_resource: vm1_port_internal
```

```
vm1_port_internal:
  properties:
    fixed_ips:
    - subnet_id:
        get_resource: internal_subnet
    network_id:
      get_resource: internal
    security_groups:
    - get_param: secgroup1
  type: OS::Neutron::Port
vm2:
  properties:
    flavor:
      get_param: flavor-2048-1-20-cirros-0.4.0-x86_64-disk
    image:
      get_param: img1
    key_name:
      get_param: key1
    name: vm2
    networks:
    - port:
        get_resource: vm2_port_internal
  type: OS::Nova::Server
vm2_port_internal:
  properties:
    fixed_ips:
    - subnet_id:
        get_resource: internal_subnet
    network_id:
      get_resource: internal
    security_groups:
    - get_param: secgroup1
  type: OS::Neutron::Port
internal_network:
  properties:
    name: internal_network
  type: OS::Neutron::Net
internal_subnet:
  properties:
    cidr:
      get_param: cidr1
    gateway_ip:
      get_param: gateway1
    network_id:
      get_resource: internal_network
  type: OS::Neutron::Subnet
```

```
  router:
    properties:
      name: external_router
      external_gateway_info:
        external_fixed_ips:
          - ip_address: 20.20.0.1
        network:
          get_param: public_net1
    type: OS::Neutron::Router
  router1_interface1:
    properties:
      router_id:
        get_resource: router1
      subnet_id:
        get_resource: internal_subnet
    type: OS::Neutron::RouterInterface
```

Figure 4: YAML template describing the minimal network scenario.

The definition of complex networks requires the inclusion in the schema of multiple networks and subnets, described by their own Classless Inter-Domain Routing (CIDR) and gateway (along with additional optional parameters).

The communication among user-defined networks will be enabled interposing specific Linux VMs, acting as routers, with port on both networks. These specific VMs will be configured with the flag *ip_forward* active, and specific *iptables* rules to enable packet forwarding.

3.2.1 Floating IPs

OpenStack gives the opportunity to administrators to associate VMs with specific *Floating IPs*. Each VM is characterized by one or more private Internet Protocol (IP) addresses, which are used for communication between the instances. Floating IPs are public addresses that can be used for communication with networks outside the cloud, including the Internet. Each OpenStack installation manages a predefined number of Floating IPs that can be associated to running VMs. They take address within the address space of the OpenStack installation and can be accessed by external actors.

The platform performs an address translation from the floating IP address to the IP of the associated port. This translation is fully transparent for the guest VM, which will manage its own usual network interfaces.

In THREAT-ARREST, the Emulation Tool associates a Floating IP to each VM that is part of the training scenario, in order to give direct access via SSH and Remote Desktop Protocol (RDP) to the VM through the Guacamole web interface (see Section 3.3).

The following code snippet shows the section in the HOT template used to associate the floating IP to a specific port using the *OS::Neutron::FloatingIP* resource type (Figure 5).

```
vm1_floating_ip:
  properties:
    floating_network_id:
      get_param: public_net1
    port_id:
      get_resource: vm1_port_internal
  type: OS::Neutron::FloatingIP
```

Figure 5: Floating IP association.

3.3 Apache Guacamole

Apache Guacamole (Apache Foundation, s.d.) is a web-based remote desktop gateway. It provides access to remote systems exploiting SSH, RDP, Virtual Network Computing (VNC) and other remote desktop protocols, allowing connection to any kind of operating systems supporting the above protocols. It is distributed under the Apache Open Source license 2.0. The Guacamole protocol provides a common entry point for different types of machine running different types of remote desktop protocols. Its architecture makes it completely transparent for the user that has only to connect to the Guacamole web page to gain access to the VMs associated to her/his profile (see Figure 6).

The Emulation Tool exploits Guacamole as the gateway for the remote access to the VMs instanced during the deployment of the training scenario. For each VM, when the deployment of the training scenario is complete, the Emulation Tool creates a Guacamole user profile and connects this profile with the specific remote machine using the communication protocol indicated in the input eXtensible Markup Language (XML) file.

Then, the Emulation Controller returns the data shown in Figure 7, containing, for each deployed VM, username and password needed to access the remote machine through the Guacamole interface.

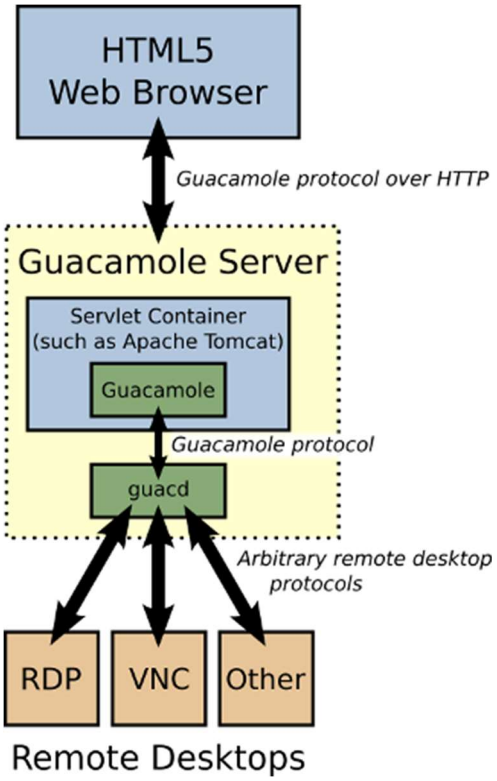

Figure 6: Guacamole Protocol (Apache Foundation, s.d.)

The Guacamole server is installed inside the Emulation Tool platform as a web application of the internal Tomcat application server. It is accessible at the server access using the port 8088 (see Figure 7 below). The THREAT-ARREST tools that have to provide access to the deployed VMs, once requested to the Emulation Controller the deployment of the training scenario, will use the returned list to notify users on the username and password to be used, or directly open the web page passing the parameters as Transmission Control Protocol (TCP) requests.

The Emulation Compiler creates one profile and the related connections for each created VM. Figure 8 shows an excerpt of the code of the Compiler that manages the insertions of users and connections to the Guacamole DB.

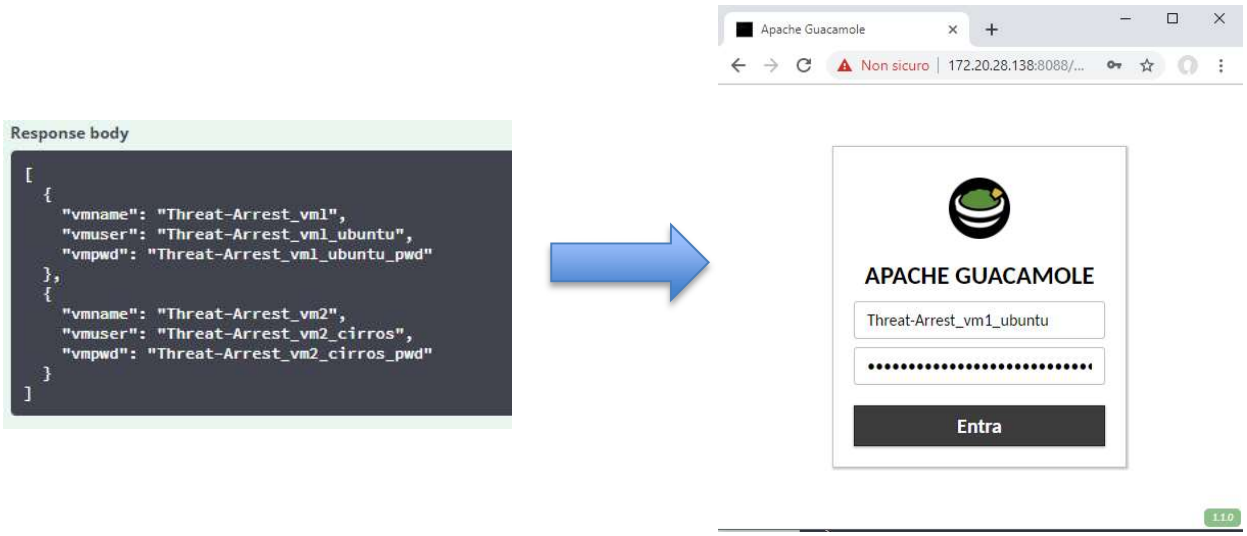

Figure 7: Remote connection credentials returned by the Emulation Controller

```
// insert user in guacamole db and related table like entity
private int insertDbUser(String uid, String pwd, String salt) {
    int id = 0;
    try{
        Connection con = getDBConnection();
        String query = "INSERT INTO guacamole_entity (name, type) VALUES (?, 'USER')";
        PreparedStatement pst = con.prepareStatement(query);
        pst.setString(1, uid);
        System.out.println("entity INSERT = " + pst.executeUpdate());

        query = "INSERT INTO guacamole_user (entity_id, password_hash, password_salt, password_date)";
        query += "',SELECT entity_id, decode('"+pwd;
        query += "', 'hex'), decode('"+salt+"', 'hex'), CURRENT_TIMESTAMP";
        query += "FROM guacamole_entity WHERE name = '"+uid;
        query += "' AND guacamole_entity.type = 'USER'";
        pst = con.prepareStatement(query);
        System.out.println("guacamole_user INSERT = " + pst.executeUpdate());
        con.close();
    } catch (SQLException ex) {
        ex.printStackTrace();
    }

    return id;
}
// create connection for guacamole_connection
private int insertConnection(String name, String protocol) {
    int id = 0;
    try{
        Connection con = getDBConnection();

        String query = "INSERT INTO guacamole_connection (connection_name, protocol) VALUES(?, ?)";
        PreparedStatement pst = con.prepareStatement(query);
        pst.setString(1, name);
        pst.setString(2, protocol);
        System.out.println("guacamole_connection INSERT = " + pst.executeUpdate());
        con.close();
    } catch (SQLException ex) {
        ex.printStackTrace();
    }

    return id;
}
// add parameter for guacamole_connection
private void insertConnectionParam(int idconn, String key, Object value) {
    try{
        Connection con = getDBConnection();
        String query = "INSERT INTO guacamole_connection_parameter VALUES (?, ?, ?)";
        PreparedStatement pst = con.prepareStatement(query);

        pst.setInt(1, idconn);
        pst.setString(2, key);
        pst.setObject(3, value);
        System.out.println("guacamole_param INSERT = " + pst.executeUpdate());
        con.close();
    } catch (SQLException ex) {
        ex.printStackTrace();
    }
}
```

Figure 8: Guacamole user and connection creation

4 Emulation Tool Modules for the Interlinking of Components

In the following sections we provide an overview of the algorithms used in the Emulation Compiler to define the networks elements inside the YAML template.

As indicated in the deliverable D2.1, the input XML file used for the initial prototype is not the final CTTP sub-model that will be defined in WP3, but an initial description of all the objects that will be deployed to instance the training scenario. In this deliverable, we describe the creation and deployment of the network only through the Emulation Compiler.

The first iteration of the Emulation Tool implementation has been focused on the setting up of the environment and the implementation of the emulated component generator, as result of the task T2.1. During the activities of task T2.3, strictly related and interconnected with the task T2.1, we focused on the designing of the algorithms for the management of the deployment of complex virtual networks, and its integration in the Emulation Tool. At the time of writing, emulated components are connected only through a simple flat network.

4.1 Emulation Compiler

This section describes the structure of the emulation compiler module and the algorithm and the rules followed to convert the input XML file into HEAT YAML files to manage the virtual network topology. The proposed algorithm is complementary with the procedures described in the deliverable D2.1 for the creation and deploying of the VM inside OpenStack.

The complete template is then passed to the emulation Engine and applied to OpenStack for the actual deployment.

4.1.1 Network Topology Definition

Network topologies in OpenStack are characterized by two main objects: *networks* and *routers*. Simplifying, each network can communicate only through a node that acts as router.

Then, each network is associated to one or more subnets to allow IP addressing and segmentation. Each subnet is characterized by a CIDR, that clearly defines the IP range, the subnet mask (e.g., 10.10.0.0/24), and a gateway.

Routers between user-defined networks are represented by special Linux VMs that act as virtual routers and enable packet forwarding. These VMs have virtual network interfaces (i.e., object of type *OS::Neutron::Port*) plugged into the two connected networks, and they have the kernel value *net.ipv4.ip_forward* set to 1 in order to enable the transmission of packets. The ports are hence the mechanism used by the emulated components to communicate through the network exploiting the common networking protocols (e.g., TCP, Ethernet, etc.). Under the point of view of a VM (i.e. emulated component), a port is seen as and it is managed like a common network interface.

Optionally, specific iptables rules can be indicated as user-data scripts to apply specific forwarding rules. It is important to note that the *OS::Neutron::Router* type of OpenStack can be associated to networks that have been declared as external, that is, to networks that are directly connected to the host network with direct access to Internet. This limitation required the use of special VMs to act as virtual routers[1].

Taking the above requirements into consideration, the input XML file has been defined in the following way (see Figure 9).

[1] More advanced solutions or adaptation of the *OS::Neutron::Router* type will be investigated in the next version of the deliverable.

The first steps related to the definition of the VMs has already been described in the deliverable D2.1. The following steps define the type and characteristics of the network topology.

As already described, the Emulation Compiler takes as input the XML file and executes the following steps to create the YAML template:

1) Retrieve the *Networks* element in the file.
2) For each *Network* in *Networks* create an object of type *OS::Neutron::Net*:
    a. use the value of the parameter *id* as name of the object.
3) For each *Network* in *Networks* create an object of type *OS::Neutron::Subnet* and collect the sub-element *cidr*:
    a. Assign to the object the name *<network name>-subnet;*
    b. Create a parameter with name equal to the value of the parameter *name* and value the string in *val*;
    c. Valorize the properties *cidr* of the object *Subnet* with the value of the related parameter *cidr* using the command *get_param;*
    d. Create a parameter with name equal to the value of the parameter *gateway* and value the string in *val*;
    e. Valorize the properties *gateway* of the object *Subnet* with the value of the related parameter *gateway_ip* using the command *get_param;*
    f. Valorize the parameter *network_id* with the related object *Net* created at step 2) with the command *get_resource;*
    g. If the parameter *is_external*has value *true*, mark the network to be connected to the common public network.
4) Retrieve the *Routers* element.
5) For each *Router* in *Routers* create an object of type *OS::Nova::Server* to act as virtual router:
    a. Use the standard *router_flavor, key* and *router_image;*
    b. Add an element in the list within the property *networks* for both networks; use the command *get_resource* applied to the string composed by <router id>-<value of *network1*> and <router id>-<value of *network2*>
    c. Create an object of type *OS::Neutron::Port* for the first connected network;
    d. Associate to the parameter *subnet-id* the object *Subnet* with the command *get_resource:* < value of *network1*>*-subnet;*
    e. Associate to the parameter *network-id* the object N*et* with the command *get_resource:* < value of *network1*>;
    f. Associate the common security group with the command *get_param: <security group>;*
    g. Create an object of type *OS::Neutron::Port* for the second connected network;
    h. Associate to the parameter *subnet-id* the object *Subnet* with the command *get_resource:* < value of *network2*>*-subnet;*
    i. Associate to the parameter *network-id* the object N*et* with the command *get_resource:* < value of *network2*>;

j. Associate the common security group with the command *get_param: <security group>*.

6) For each network marked as external:

a. Create an object of type *OS::Neutron::Router* with name *external-router[i];*

b. Associate the common external network assigning to the property network the value of the parameter indicating the public network, using the command *get_param:<name of public network>;*

c. Create an object of type *OS::Neutron::RouterInterface* with name *external-router[i]-interface*

d. Assign the name to the property *router_id;*

e. Associate the subnet related to the marked network at the property *subnet_id* with the command *get_resource.*

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<Scenario name="Threat-Arrest">
    <CustomVM name="vm1">
        <connectionmode port="22" connectiontype="ssh"></connectionmode>
        <ram val="2048"></ram>
        <vcpus val="1"></vcpus>
        <disk val="80"></disk>
        <image name="img1" val="Ubuntu-16.04-LTS-Xenial-Xerus"></image>
        <Network idref="internal" />
    </CustomVM>
    <CustomVM name="vm2">
        <connectionmode port="22" connectiontype="ssh"></connectionmode>
        <ram val="2048"></ram>
        <vcpus val="1"></vcpus>
        <disk val="20"></disk>
        <image name="img2" val="cirros-0.4.0-x86_64-disk"></image>
        <Network idref="dmz" />
    </CustomVM>
    <Networks>
        <Network id="internal" >
            <gateway name="gateway1" val="10.10.10.1"></gateway>
            <cidr name="cidr1" val="10.10.10.0/24"></cidr>
            <is_external val="false"></is_external>
        </Network>
        <Network id="dmz" >
            <gateway name="gateway2" val="20.20.20.1"></gateway>
            <cidr name="cidr2" val="20.20.20.0/24"></cidr>
            <is_external val="true"></is_external>
        </Network>
    </Networks>
    <Routers>
        <Router id="router1">
            <routerInterface network_1="dmz" network_2="internal"></routerInterface>
        </Router>
    </Routers>
</Scenario>
```

Figure 9: Input XML for complex networks.

4.1.2 Results

The application of the algorithm above led to the production of the YAML template in Figure 11. The results are also shown in Figure 10, taken from the OpenStack Dashboard administration interface.

In this image, it is possible to see clearly the user-defined networks (orange and green), the VM acting as router between them, and the router connecting the demilitarized zone (DMZ) – special local network configuration type – with the common public network (blue).

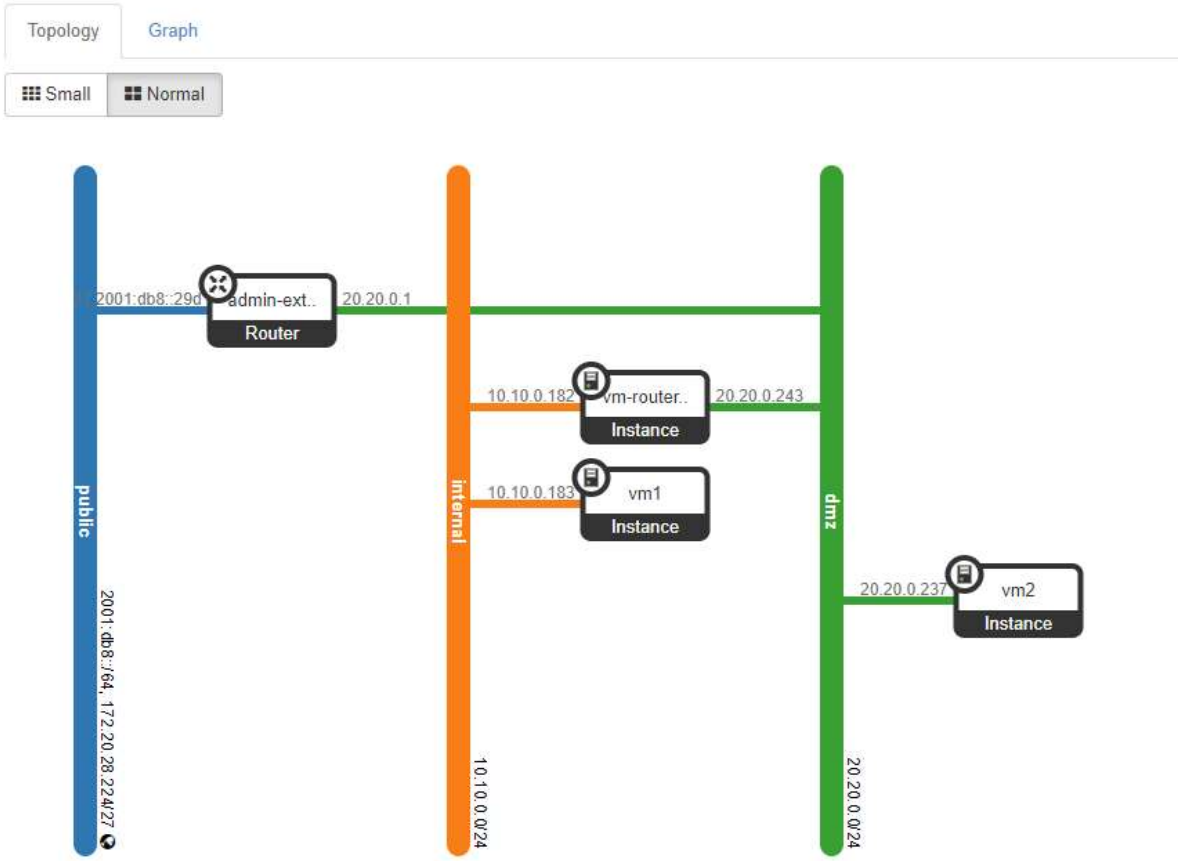

Figure 10: User-defined network and VMs connection shown by OpenStack interface.

```
description: 'Stack of scenario Threatarrest '
heat_template_version: '2018-08-31'
parameters:
  cidr-internal:
    default: 10.10.0.0/24
    type: string
  cidr-dmz:
    default: 20.20.0.0/24
    type: string
  flavor-2048-1-20-cirros-0.4.0-x86_64-disk:
    default: flavor-2048-1-20-cirros-0.4.0-x86_64-disk
    type: string
  flavor-router:
    default: flavor-2048-2-10-Ubuntu-16.04-LTS-Xenial-Xerus
    type: string
  gateway-internal:
    default: 10.10.0.1
    type: string
  gateway-dmz:
    default: 20.20.0.1
    type: string
  img-router:
    default: Ubuntu-16.04-LTS-Xenial-Xerus
    type: string
  img:
    default: cirros-0.4.0-x86_64-disk
    type: string
  key1:
    default: common-key
    type: string
  public_net1:
    default: 0ec0b875-0a61-460e-be46-8214cd9a6d5c
    type: string
  secgroup1:
    default: f9a3fe5a-1d5b-40e6-b627-f0a9a1f57ce6
    type: string
```

```
resources:
  vm1:
    properties:
      flavor:
        get_param: flavor-2048-1-20-cirros-0.4.0-x86_64-disk
      image:
        get_param: img
      key_name:
        get_param: key1
      name: vm1
      networks:
      - port:
          get_resource: vm1_port_internal
    type: OS::Nova::Server
  vm1_port_internal:
    properties:
      fixed_ips:
      - subnet_id:
          get_resource: internal-subnet
      network_id:
        get_resource: internal
      security_groups:
      - get_param: secgroup1
    type: OS::Neutron::Port
  vm2:
    properties:
      flavor:
        get_param: flavor-2048-1-20-cirros-0.4.0-x86_64-disk
      image:
        get_param: img
      key_name:
        get_param: key1
      name: vm2
      networks:
      - port:
          get_resource: vm2_port_dmz
    type: OS::Nova::Server
  vm2_port_dmz:
    properties:
      fixed_ips:
      - subnet_id:
          get_resource: dmz-subnet
      network_id:
        get_resource: dmz
      security_groups:
      - get_param: secgroup1
    type: OS::Neutron::Port
```

```
internal:
  properties:
    name: internal
  type: OS::Neutron::Net
internal-subnet:
  properties:
    cidr:
      get_param: cidr-internal
    gateway_ip:
      get_param: gateway-internal
    network_id:
      get_resource: internal
  type: OS::Neutron::Subnet
dmz:
  properties:
    name: dmz
  type: OS::Neutron::Net
dmz-subnet:
  properties:
    cidr:
      get_param: cidr-dmz
    gateway_ip:
      get_param: gateway-dmz
    network_id:
      get_resource: dmz
  type: OS::Neutron::Subnet
vm-router1:
  properties:
    flavor:
      get_param: flavor-router
    image:
      get_param: img-router
    key_name:
      get_param: key1
    name: vm-router1
    networks:
    - port:
        get_resource: vm-router_port_dmz
    - port:
        get_resource: vm-router_port_internal
  type: OS::Nova::Server
```

```
vm-router_port_dmz:
  properties:
    fixed_ips:
    - subnet_id:
        get_resource: dmz-subnet
    network_id:
      get_resource: dmz
    security_groups:
    - get_param: secgroup1
  type: OS::Neutron::Port
vm-router_port_internal:
  properties:
    fixed_ips:
    - subnet_id:
        get_resource: internal-subnet
    network_id:
      get_resource: internal
    security_groups:
    - get_param: secgroup1
  type: OS::Neutron::Port
external-router1-interface:
  properties:
    router_id:
      get_resource: external-router1
    subnet_id:
      get_resource: dmz-subnet
  type: OS::Neutron::RouterInterface
external-router1:
  properties:
    external_gateway_info:
      network:
        get_param: public_net1
  type: OS::Neutron::Router
```

Figure 11: YAML template related to the input XML in Figure 9.

The algorithm can easily accommodate more complex topologies with several networks.

5 Conclusions

In this deliverable, we provide a description of the algorithms and software for the definition of complex networks inside OpenStack, to be used for the deployment of THREAT-ARREST training scenario.

The deliverable is the results of the first iteration of task T2.3 and paves the way for the next iteration and for the final version of the Emulation Tool due at M24. The flexibility and modularity of the OpenStack network types give to developers the possibility to include in the topology also real devices as IoT virtualized objects, in order to deploy training scenario based on real-world devices (Merlino, et al., 2014).

The virtual network deployed through the input XML will allows all the emulated components to establish and manage connections with each other, within the same virtual subnet.

References

[1] Apache Foundation, s.d. *Apache Guacamole.* [Online]
Available at: https://guacamole.apache.org/
[Consultato il giorno 2019].

[2] Cesena, M., et al. 2017. SHIELD Technology Demonstrators. CRC Press, Book for Measurable and Composable Security, Privacy, and Dependability for Cyberphysical Systems, pp. 381-434.

[3] Ferrera, E., et al. 2018. IoT European Security and Privacy Projects: Integration, Architectures and Interoperability. CRIStin – SINTEF, Next Generation Internet of Things. Distributed Intelligence at the Edge and Human Machine-to-Machine Cooperation. Book Chapter 7, pp. 207-292.

[4] Hatzivasilis, G., et al., 2017. SecRoute: End-to-End Secure Communications for Wireless Ad-hoc Networks. 22nd IEEE Symposium on Computers and Communications (ISCC 2017), IEEE, Heraklion, Crete, Greece, 03-06 July 2017, pp. 558-563.

[5] Hatzivasilis, G., et al., 2019. MobileTrust: Secure Knowledge Integration in VANETs. ACM Transactions on Cyber-Physical Systems – Special Issue on User-Centric Security and Safety for Cyber-Physical Systems, ACM, vol. 4, issue 3, Article no. 33, pp., March 2020.

[6] Merlino, G. et al., 2014. *Stack4Things: Integrating IoT with OpenStack in a Smart City context.* Hong Kong, China, s.n.

[7] OpenStack, 2019. *OpenStack Compute (Nova).* [Online]
Available at: https://docs.openstack.org/neutron/latest/
[Accessed 2019].

[8] SimPlan AG, 2019. *Discrete-event simulation library jasima.* [Online]
Available at: https://www.simplan.de/en/software-2/jasima/
[Accessed 2019].

Appendix I – HEAT template for the smart vessel

The following HOT implements the smart vessel infrastructure, as described in Section 2.

```
heat_template_version: 2019-06-21
description: HOT template for two interconnected VMs (captain's PC and on-deck equipment)

parameters:
  image_id:
      type: string
      description: Image Name
  secgroup_id:
      type: string
      description : Id of the security group
  public_net:
      type: string
      description: public network id (external OpenStack connection)

resources:
  private_net:
      type: OS::Neutron::Net
      properties:
        name: private-net
  private_subnet:
      type: OS::Neutron::Subnet
      properties:
        network_id: { get_resource: private_net }
        cidr: 172.16.2.0/24
        gateway_ip: 172.16.2.1
  router1:
      type: OS::Neutron::Router
      properties:
        external_gateway_info:
            network: { get_param: public_net }
  router1_interface:
      type: OS::Neutron::RouterInterface
      properties:
        router_id: { get_resource: router1 }
        subnet_id: { get_resource: private_subnet }

  VM1_port:
      type: OS::Neutron::Port
      properties:
        network_id: { get_resource: private_net }
        security_groups: [ get_param: secgroup_id ]
        fixed_ips:
            - subnet_id: { get_resource: private_subnet }
  VM1_floating_ip:
      type: OS::Neutron::FloatingIP
      properties:
        floating_network_id: { get_param: public_net }
        port_id: { get_resource: VM1_port }
  VM1:
      type: OS::Heat::SoftwareConfig
      properties:
        name: captain_pc
        image: windows_10_pro_iso_64_bit_iso_october_2018_update_v_1809
        flavor: m1.xlarge
        networks:
            - port: { get_resource: VM1_port }
```

```
  VM2_port:
      type: OS::Neutron::Port
      properties:
        network_id: { get_resource: private_net }
        security_groups: [ get_param: secgroup_id ]
        fixed_ips:
            - subnet_id: { get_resource: private_subnet }
  VM2:
      type: OS::Heat::SoftwareConfig
      properties:
        name: On-Deck_Equipment
        image: Ubuntu-Jasima
        flavor: m1.xlarge
        networks:
            - port: { get_resource: VM2_port }

outputs:
  VM1_private_ip:
      description: Private IP address of captain's PC
      value: { get_attr: [VM1, first_address ] }
  VM1_public_ip:
      description: Floating IP address of captain's PC
      value: { get_attr: [VM1_floating_ip, floating_ip_address ] }
  VM2_private_ip:
      description: Private IP address of on-deck equipment
      value: { get_attr: [VM2, first_address ] }
```