Security considerations in Docker Swarm networking

Marcel Brouwers [*]

UNIVERSITY OF AMSTERDAM
MASTER THESIS PROJECT, SYSTEM AND NETWORK ENGINEERING

July 28, 2017

Abstract

When using overlay networks it can get complicated to keep track of the structure of the networks as well as keeping track of what gets exposed on these networks. This master thesis project focuses on testing what can be done on an overlay network if a Docker container that is part of a Docker Swarm overlay network gets compromised and on visualizing the overlay networks and what they expose. When running Docker Swarm in default configuration it was not possible to perform layer 2 attacks that rely on injecting ARP packets on the network. Injecting packets in the VXLAN tunnels of the overlay network did turn out to be possible in certain cases. Visualizations of the overlay networks have been generated using the Docker API and the D3 framework in order to show the structure of the overlay networks and to show ports opened on containers.

[*]Special thanks to Esan Wit

Contents

1 Introduction

With a large number of virtual hosts in a network it can get complicated to keep track of what these hosts expose on the internal network and to external networks. With the trend of containers to run services, the number of hosts might grow even further and keeping track of what is exposed becomes more challenging. A popular solution for using containers comes from the Docker project. Docker enables automated deployment of software within containers. The Docker project introduced Docker Swarm which allows for central deployment of containers on multiple nodes. Docker Swarm has a feature for VXLAN [1] based overlay networks between containers. With this feature containers can easily be added to the same virtual layer 2 domain. Now not only the existing network has to be taken into account in order to keep track of what gets exposed but overlay networks become part of the challenge. This master thesis project looks into the security of Docker Swarm overlay networks and visualizing the security boundaries in the overlay network. A security boundary is a boundary within or at the edges of a system or network under a single administrative control. For example, network segments or firewalls.

The research question is defined as follows:

What gets exposed when using Docker Swarm overlay networks and is there a way to visualize what gets exposed?

In order to answer the main research question, the following sub-questions are defined:

1. Which security measures are there for Docker Swarm overlay networks and what can be done on the overlay network if a container or host gets compromised?
2. Which strategies are there to find out what gets exposed by containers and hosts in (overlay) networks and how effective are they?
3. Is it feasible to consolidate all the information about exposure and visualize it in a comprehensible way?

1.1 Motivation

As mentioned previously, keeping track of security boundaries can get complicated with the introduction of overlay networks. When administering the network of an organization it can be challenging to keep track of security boundaries on the network and with the addition of virtual networks the challenge grows. Since being able to identify security boundaries in a network is of importance to ensure adequate security, solutions have to be looked at.

1.2 Container security

There is plenty of discussion on security for container solutions. Some argue that container solutions should not be used for running untrusted applications since the same kernel is shared between the containers and that containers should only be used for dependency untangling and simpler deployments [2][3]. The Docker project states the following about container security: "Docker containers are, by default, quite secure; especially if you take care of running your processes inside the containers as non-privileged users" [4]. Since container security is a discussion on its own the focus in this research will be limited to the Docker Swarm overlay networks and visualizing the networks and security boundaries.

2 Related Work

In the work performed by Peneda [5] the feasibility of layer 2 attacks on a VXLAN overlay network was investigated. In the research the VXLAN implementation of the Linux kernel was tested. It was concluded that ARP attacks are possible in the VXLAN overlay networks tested in the research. An attack with nested VXLAN packets was also tested but was not successful.

Research into Docker overlay networks was performed by Hermans and de Niet [6]. In this research the performance of different Docker overlay network solutions was tested. The conclusion was that overlay solutions performed similarly to non-overlay solutions in the area of latency and jitter. With regard to throughput irregular results were found. Other research in the performance of Docker overlay networks was performed by Zismer [7]. This research was also aimed at the performance of different Docker overlay network solutions and it was concluded that the performance is heavily dependent on the CPU performance of the machine. In both researches the Docker Swarm overlay network solution was not tested.

With regard to the security of overlay networks a document by NIST [8] about secure virtual network configuration was published. One of the recommendations is to isolate the traffic from VXLAN networks on the physical network using VLAN-like techniques. It is also recommended that overlay virtual network deployments use centralized or federated SDN controllers. The Docker documentation [9][10] states that the nodes exchange information about the overlay networks using a gossip protocol, the Raft Consensus protocol. Other sources [11] point out that for exchanging container MAC addresses between hosts a gossip protocol named Serf is being used. When looking at the source code [12] of the overlay driver from Libnetwork used by Docker we can clearly see references to the use of Serf. The documentation [9][10] also shows that the swarm nodes use encryption keys from the manager nodes to encrypt the gossip communication and data store such that containers outside of the swarm cannot attach to the overlay network.

As for visualization tools for Docker Swarm there is a project available on Github [13] "Docker Swarm Visualizer" which is able to show the user on which hosts containers are running. The tool does not give any insight in the overlay networks. Another tool available with regarding visualization of Docker Swarm solutions is "Dvizz - A Docker Swarm Visualizer" [14]. This tool shows a force-directed graph of one of the hosts in the Swarm from a service point of view. The graph shows which tasks resulting from a service are running on a host. Both tools do not show any information concerning the overlay networks within the Swarm.

3 Theory

3.1 Docker Swarm

Docker Swarm is a mode of Docker that makes it easier to manage a cluster of Docker engines. A cluster of Docker Engines, managed using swarm mode, is called 'Swarm'. On the Swarm services can be run and the Swarm will divide the containers running as part of the service over the Docker Engines that take part in the Swarm. To make the networking between containers of a service simpler, a network solution was introduced as part of the Swarm mode. The network solution is a VXLAN based overlay layer 2 domain between the containers such that the containers of a service are part of the same virtual layer 2 network even if the Docker Engines running the containers do not share a layer 2 network.

3.2 VXLAN and Swarm

The Docker Swarm overlay functionality is an implementation of the VXLAN standard [1]. With VXLAN, tunnels between hosts can be created to send packets. VXLAN is a layer 2 tunneling protocol on top of layer 3. Layer 2 packets get wrapped inside a layer 3 packet. This creates the effect that two interfaces on remote machines can be part of the same layer 2 network which can be useful in some scenarios where applications are required to be in the same layer 2 network. On devices that are in the network so called "VTEP" (VXLAN Tunnel End Point) devices are present. These are virtual devices which are endpoints of the VXLAN tunnel. Each VXLAN frame contains a VNI which is an identifier for the VXLAN. This VNI in combination with the correct port number is also the way to determine if a packet belongs to a certain VTEP. The default port used for VXLAN traffic is UDP port 4789 which is assigned [15] by IANA for the use of VXLAN.

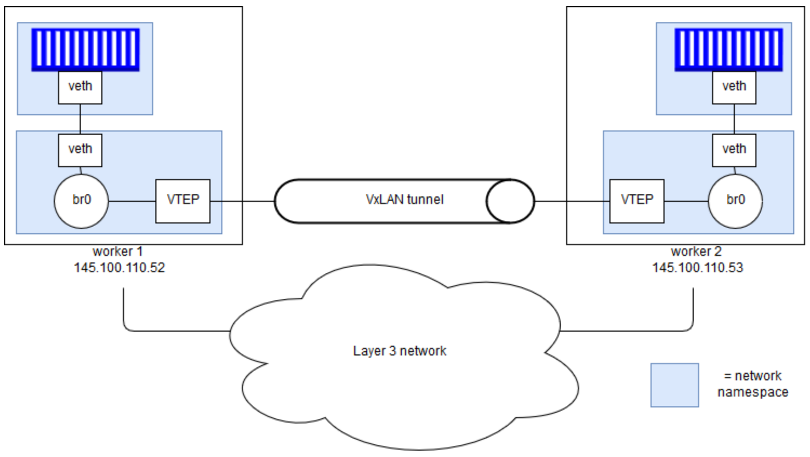

Figure 1: *VXLAN tunnel between two of the hosts in the swarm*

On a Docker host the VTEP is connected to a bridge which resides in a virtual network namespace. A network namespace is in essence another copy of the network stack with its own devices, firewall rules, arp table etc.. Each container has its own network namespace on the host. The network interface that is available inside the container is connected to the bridge using a VETH pair of which one interface of the pair resides in the namespace of the container and the other interface resides in the namespace for the VTEP which is connected to the bridge. With VXLAN as specified in RFC 7348 [1] the MAC to VTEP's IP address mapping is achieved using source-address learning. The VTEPs are joined to the same IP multicast group and the multicast address is used for sending unknown destination traffic and broadcast layer 2 frames. For Docker Swarm overlay networks this process was found to be different. This is elaborated on in the discussion section of this paper.

3.3 Security for Docker Swarm

By default when creating a Swarm overlay network the network traffic does not get encrypted. There is, however, the option to encrypt the traffic when the "–opt encrypted" flag gets specified when creating the overlay network [16]. The traffic between the nodes is then encrypted using IPsec tunnels with keys that are rotated every 12 hours. For updating the forwarding database Docker Swarm uses a gossip based protocol named Serf [17].

4 Methodology

4.1 Environment

For the experimental environment 5 Ubuntu 16.04 virtual machines were set up in a virtual environment using Xen [18]. All virtual machines were connected to the same virtual network bridge on the machine running the Xen hypervisor. Then Docker (version 17.03.1-ce) was set up on the virtual machines. There was no need to install Swarmkit separately since all the Swarm functionality came bundled in this version of Docker. Two of the virtual machines were made Docker Swarm manager nodes and three of the virtual machines were made Docker Swarm worker nodes. Subsequently the Swarm overlay networks were created from the manager nodes. Services were created on the Swarm, based on Ubuntu 16.04 images and Busybox images, and were added to the overlay networks at the time of creation.

4.2 Experiments

In order to find out what is possible on the overlay network once a container is compromised it is good to test which well known layer 2 attacks work on a VXLAN based Docker Swarm overlay network.

4.2.1 ARP spoofing

A well-known attack for classic layer 2 networks is ARP spoofing [19]. In this attack one of the containers will attempt to send out ARP packets on the network stating its MAC address belongs to a certain IP address. The hypothesis is that packets destined for the IP address will end up at another node, for example the node performing the attack. In this way a host can perform a man in the middle attack on a traditional layer 2 network (without additional security measures). The experiment can be performed using the Ettercap [20] tool which is a tool designed for man in the middle attacks based on ARP spoofing.

4.2.2 MAC flooding

MAC flooding is based on flooding a network with MAC addresses causing a switch to start acting like a hub because of an overflow of the CAM table. The CAM table is used by the switch to keep track of which MAC addresses are located at which ports. Overflowing the CAM table causes some switches to start acting like a hub forwarding all traffic to all ports and thus enabling to eavesdrop all traffic from a connection on one of the interfaces. The purpose of this attack on the VXLAN would be to test if there would be similar behaviour to this on the Swarm Overlay network. The experiment can be performed using the tool "Macof" from the Dsniff tools [21]. The ARP tables and forward databases for the different interfaces on the hosts can then be monitored for any change. Additionally it can be checked which traffic arrives at the network interface in the container to check if similar behaviour to flooding a layer 2 switch can be observed.

4.2.3 Injecting packets in the VXLAN tunnel

During the research the question was raised if it is possible to inject packets inside a VXLAN tunnel between two containers from a node that had no part in the overlay network. With overlay networks issues could arise if packets can be injected from a different layer 3 network into the layer 2 tunnel. This concern is also raised by the writers of RFC 7348 in the security considerations section [1]. In order to test this it would be easiest to capture a packet on one of the nodes the VXLAN tunnel passes and replay the packet from another node. Then alter the packet and change the source address to a node that does not take part in the overlay network and check if that works.

4.2.4 Replaying encrypted VXLAN traffic

Since Docker Swarm has a feature for encrypting the traffic on the overlay networks the question arose what would be possible with this encrypted traffic. A simple thing to test would be a replay of data to test if there is any replay protection. According to the Docker documentation [9] IPSEC tunnels are used when the overlay network is created as an encrypted overlay network. For encryption of the IPSEC tunnel the AES algorithm in GCM mode is used and the key is rotated every 12 hours [9].

4.2.5 Adding an interface to the VXLAN

As one of the experiments it is interesting to see if it is possible to add an interface to the VXLAN overlay network. Most interesting would be if an interface could be added to the network on a node that has no containers that are in the VXLAN overlay network. Also for hosts that have containers on the network it would be interesting to see if an interface of the host could be added to the VXLAN overlay network.

4.2.6 Strategies to check what gets exposed

With a traditional Docker set-up the user specifies which ports on the host machine get exposed to the Docker container. In this situation the user is aware that only the ports specified get exposed to the network. In the case of Docker Swarm overlay networks this is different. On a Docker Swarm overlay network every port of the container gets exposed to the overlay networks unless additional firewall rules are set inside the container.

For security purposes it is good to know on which containers services are running that expose ports. In a non-swarm setup the exposed ports can be requested using the Docker API. In the case of Docker Swarm this can also be done for the ports that get published to the outside of the swarm. However, inside the overlay network it is unknown what ports are open on the overlay network. Although the overlay network is normally not directly accessible, the network will still be accessible for an attacker if one of the containers gets compromised.

In general there would be two ways of finding what is exposed on an overlay network. The first method would be to perform a portscan from one of the containers inside the overlay network. One could think of having a tool like the Nmap[22] scanner running on one of the containers in the overlay network and scanning the containers connected to the overlay network.

The other method would be to have each container report the state of ports that services are listening on to a central datastore. The implication of this would be to have a script running on each container actively reporting the status of ports. Perhaps it would also be possible to have the script running on the host machine to avoid having to run scripts in every docker container. The downside of this approach would be that the state of the firewall in the virtual network namespaces would be unknown. The collection of the data, however, could be faster than using portscans.

5 Results

5.1 ARP spoofing

The ARP spoofing attack was initially attempted with the Ettercap tool. This tool, however, was unable to run inside the Docker Swarm containers due to restrictions in capabilities within the containers (for Docker Swarm containers there are currently no possibilities to extend these restrictions in capabilities [23]). This resulted in performing the attack using the tool Arpspoof from the Dsniff tools [21]. This tool seemed to run fine despite the restrictions. The ARP caches on the container of which the ARP cache should show changes was monitored but no change in the ARP table could be observed. In order to find out why this resulted in no change of the ARP table a packet capture was done on the bridge interface to which all the hosts are connected. This packet capture showed that the ARP messages that would have been wrapped inside VXLAN packets did not leave the host on which the attacking container was running. In order to verify these results are not caused by the lack of privileges inside the container an overlay network was created with the "attachable" option set. In this way the network could be attached to a container running in privileged mode and the experiment could be re-run. The result were the same. The ARP packets did not leave the host. The most likely reason for this is elaborated on in the discussion section of this document.

5.2 MAC flooding

The MAC flooding was performed from within a Docker container connected to an overlay network. The tool used for generating the packets was the "Macof" tool from the Dsniff toolset. The experiment was performed from a non-privileged container and from a privileged container connected to the overlay network. Forward databases on the VTEPs were checked during the experiment as well as the traffic received within the container on the VXLAN. No change of behaviour, like flooding all traffic to all interfaces, could be observed.

5.3 Injecting packets in the VXLAN tunnel

In this experiment ICMP packets were sent from one container to another container over the Swarm VXLAN tunnel. The (encapsulated) traffic was captured on a network interface of a host running the Docker engine. This capture was then replayed from a virtual machine that is not participating in the Swarm nor running the Docker engine. While replaying the traffic Tcpdump was running on the two containers. It could be observed that the ICMP request was received by one of the containers and a reply on this ICMP request was sent to the other container over the VXLAN tunnel.

Then the packet capture was adapted in such a way that the source IP address of the VXLAN packet was the address of a host in the swarm that was not hosting one of the containers. This adapted packet capture was then replayed and the same results could be observed. The result being that the ICMP request arrived at one of the containers and consequently an ICMP reply was received at the other container as a reply to the ICMP request.

Lastly the source IP of the VXLAN packet was adapted again in such a way that the source IP address belonged to a virtual machine not participating in the swarm nor running the Docker engine. When sending this packet the ICMP request did not arrive at the containers. This seems to indicate that the VTEP interfaces are not accepting traffic from any source.

5.4 Replaying encrypted VXLAN traffic

A Docker Swarm encrypted overlay network was created by specifying "–opt encrypted" when creating the overlay network. A service was added to the overlay network with two containers on different hosts. An ICMP ping was started in one of the containers to the other container. The traffic was captured using Tcpdump on the interface of the host with the receiving container. In the receiving container Tcpdump was also started to show the ICMP packets arriving at one of the network interfaces of the container. Then the capture on the interface of the host was stopped and the packet capture was replayed from another VM which is not part of the Docker Swarm. The result of this replay was that the ICMP packets still arrived in the container. Hence the IPSEC tunnel for encrypted Docker Swarm overlay networks seems to be vulnerable to replay attacks within the 12 hours in which the key is not rotated.

5.5 Adding an interface to a VXLAN

If the VXLAN is created with the attachable flag it is possible to add regular docker containers to the Swarm overlay network. However, it would be interesting if it is possible to create a usable interface in the default network namespace on a Docker host machine or even more so if a VTEP could be added on a remote machine that has access to the overlay network. Both of these were tried in this experiment. A VETH pair was created in the default network namespace. Then one interface of the pair was moved inside the network namespace of a container and connected to the bridge. The interface in the default network namespace was then given an IP address within the overlay network. The result was that the interface was able to communicate with the interface that is part of the overlay network on the same Docker Engine. However, it was not possible to achieve full communication with other containers on other hosts in the overlay network. The IP address of the newly added interface did show up in the ARP table of the other containers on other hosts but the entry did not contain a MAC address.

An attempt was also done adding a VTEP interface on a host that is not part of the Swarm. It is possible to specify the same port number the VTEP is listening on and the same VNI for the interface. An IP multicast group could not be specified when creating the VTEP since Swarm overlay networks do not rely on an IP multicast group for sending out packets with an unknown destination. Because there is no multicast group the newly created VTEP interface was not able to learn the MAC to IP mappings for the forward database. In addition to this, the VTEP interfaces on the other hosts in the Swarm were not able to learn about the newly created interface.

5.6 Strategies to check what gets exposed

The first strategy tested to check what gets exposed is by plotting the networks that connect the nodes. This was done using the D3 plus framework[24] with data coming from the Docker API[25] running on one of the manager nodes. One of the examples of D3plus[26] was used as a basis for creating a diagram showing the different VXLANs as points with containers connected to it. This can be seen in figure 2.

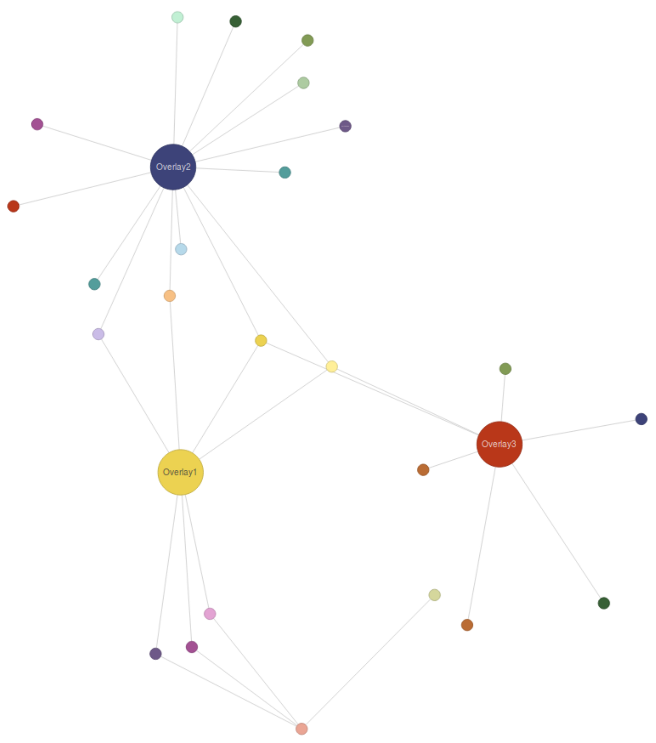

Figure 2: *Plot showing the VXLANs and connected containers*

Which containers are connected in the same VXLAN can also be shown using an edge bundling diagram. This diagram connects the container IDs which share a VXLAN together. For small environments this works to see which containers are connected in the same VXLAN overlay network. However, for a large environment it can be difficult to keep the overview with such a diagram. That is why the visualization was then adapted to show the overlay networks as nodes in a circle, making the network paths simpler to see, like is shown in figure 3. The visualization in the figure is based on Mike Bostock's Edge Bundling diagram[27] adapted to use information from the Docker API. The PHP script for gathering the data for this visualization is showin in listing 9 of the appendix.

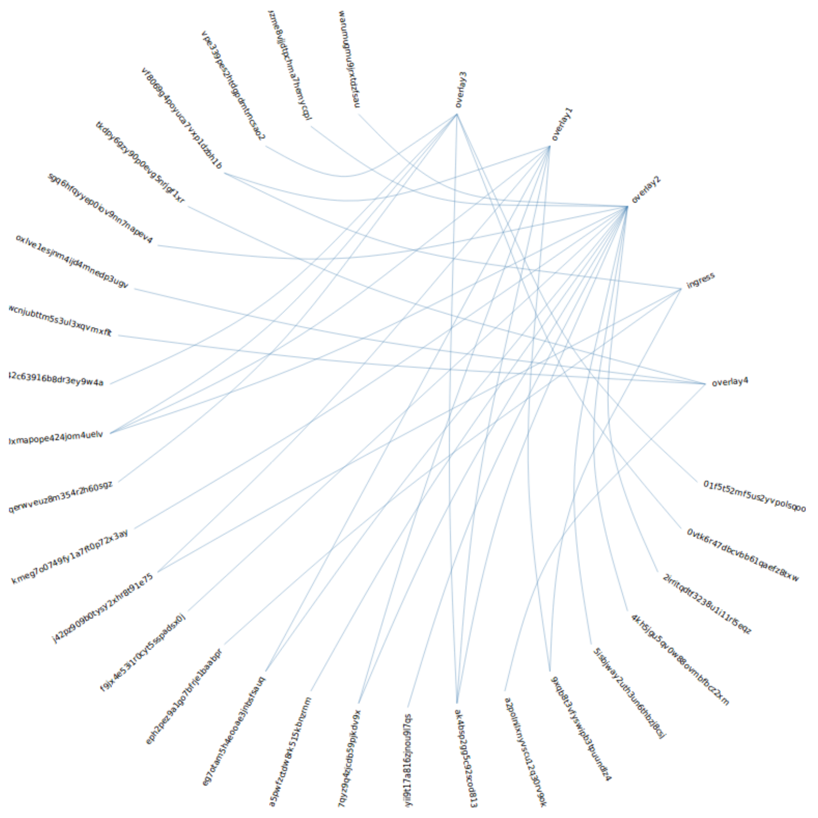

Figure 3: *Plot showing the VXLANs and the containers connected to them in an Edge Bundeling diagram.*

The plots shown are useful for understanding which containers have interfaces in which overlay networks. Since a separation of layer 2 networks is a security boundary, one could say that the figure helps in understanding the security boundaries that are a result of having different overlay networks. Another strategy is looking at open ports on containers in a network. Using a scanner like Nmap on one of the containers in the overlay network, a list of open ports can be gathered. A Python script was created that starts scanning the subnets of connected interfaces and posts the results of the scan in a JSON format to a webserver. The script is shown in listing 1 of the appendix. The advantage is that this would most likely give a good indication of what a possible attacker might be able to find on an overlay network. The downside is that for each overlay network a container would have to be set up to do the scanning. Additionally scanning a network may take a long time depending on the type of scan. The script is made available in a Github repository [28].

Figure 4 shows a visualization where the containers are visualized as nodes connected to nodes that represent the VXLANs, similar to what is shown in figure 2. The difference is that listening ports on the nodes are also represented as nodes in the network and are connected to the nodes that represent the containers. The collection of the data used in this graph is done by a Python script which runs on the host which runs the Docker Engine. The script requests the netstat output for the different network namespaces on the docker engine host and sends them to a server running PHP and MySQL. This Python script is shown in listing 2 of the appendix. The PHP scripts on the receiving end on the server are shown in listing 5 and listing 6 of the appendix. The result is a central database containing the listening ports for the containers in the Swarm cluster without using a port scanner like Nmap. The PHP and HTML scripts to generate the data for figure 4 can be found in listing 7 and listing 8 of the appendix. The database structure is defined in appendix 4 of the appendix.

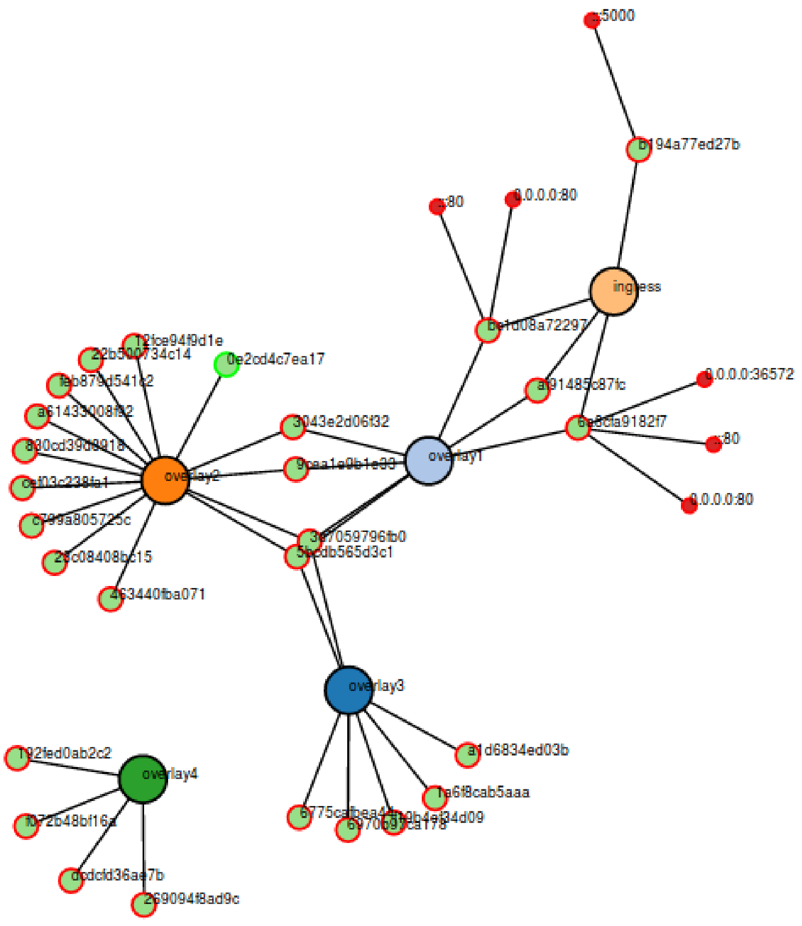

Figure 4: *Visualization showing the containers connected to the overlay networks. Each open port is represented by a red node connected to the container.*

Finally, it might be of interest to easily view which containers are running on which node. Figure 5 on the following page shows a visualization displaying which containers run on which node. This way, when a container gets compromised and broken out of, one might be able to assess which other containers might be affected because they run on the same host. The visualization is based on Mike Bostock's Circle Packing diagram [29] and the data is generated using a PHP script parsing information from the Docker API. In this case the host running the Docker Engine could be considered a security boundary. The PHP script for collecting the data for this visualization is shown in listing 10 of the appendix. A visualization showing which container runs on which node was also available from the Docker Swarm Visualizer on Github [13] visualizing the nodes and containers.

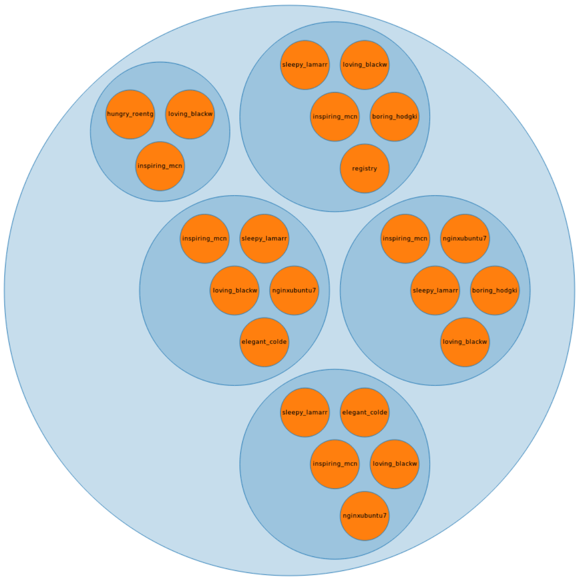

Figure 5: *Figure showing which containers run on which Docker Engine*

5.7 Consolidating information

Since it is difficult to consolidate all the relevant data in a single diagram that is understandable, the next most logical step is to have multiple diagrams available in the same location. A system administrator could then use the diagrams to get a better understanding of the overlay networks, which nodes are part of the networks and which ports are in a listening state on which containers. Therefore creating some sort of a dashboard displaying the relevant information is a logical step.

The result of this is that the diagrams shown above are collected in a single web interface such that Docker Swarm administrators have a slightly better overview of their Docker Swarm overlay networks. The host running the web interface has to be able to interact with the Docker API on one of the manager nodes and should be able to host a database for storing the results from the scripts running on the hosts.

6 Discussion

When testing the layer 2 attacks on the Docker Swarm overlay networks it turned out to be difficult to run the tools inside the containers created in Swarm mode due to containers having limited privileges. In a non-Swarm environment this can be solved by starting the containers with the "–privileged" flag in order to give the container more permissions. For containers started with Docker Swarm this is currently not possible (yet). However it is possible to create the overlay network using the "–attachable" flag which makes the overlay network attachable to containers that are created using the "docker run" command on machines that are part of the Swarm. This is what was done in this research in order to make sure a container could be started in privileged mode to run the Ettercap tool. The packets resulting from the ARP injection did not arrive at the Docker Engine that ran the other container that was the target of the attack. After further investigation the cause of this seems to be that Docker Swarm overlay networks do not make use of IP multicast groups for sending unknown ARP packets to VTEP interfaces in the overlay network. Instead the forward databases of the VTEPs are populated by the Docker Engine. For this Docker makes use of the gossip protocol 'Serf' to fill these forward databases. The ARP traffic is taken care of using a technique named 'ARP proxy'. When checking the VTEP interfaces within the virtual namespaces on the Docker hosts it was confirmed that ARP proxy was indeed set on these interfaces. Hence no ARP packets could be found when sniffing the VXLAN tunnel.

Injecting packets in the VXLAN tunnel turned out to be possible if the source address of another docker engine which is part of the Swarm is used as the sender IP of the VXLAN packet. In the test an ICMP request was sent in the tunnel and the reply by the container was sent to another node in the overlay network. In this test it was not possible to have a session build up by injecting packets in the VXLAN tunnel making the real world usefulness of the attack questionable. Perhaps in rare cases with protocols relying on UDP packets some sort of attack might be possible. During this experiment it was also noted that the VXLANs that are created use VNIs which are generated in sequential order, starting at 4096. The destination port for the traffic is by default port 4789 as per RFC 7348 [1].

It has proven to be difficult to add a fully functional interface to the VXLAN that is not part of a Docker Container. Although it was possible to connect to an interface to a bridge interface that is part of the VXLAN, the IP address of the interface did not end up in the forwarding tables on the other machines resulting in one-way communication with containers on the VXLAN running on other hosts and two-way communication with containers running on the same host.

What stands out when comparing the results of the experiments with the experiments from Peneda [5] is that the attempts of ARP injection on the Swarm overlay network are not successful while in the research performed by Peneda [5] these attempts were successful. This difference can be explained by the fact that in the research of Peneda the VTEP interfaces make use of IP multicast groups to fill the forward database of the VTEP. In the case of Docker Swarm the Serf gossip protocol is used to fill the forward databases instead of an IP multicast group.

Using data from both the Docker API and data from scripts running on the hosts which run the Docker Engine visualizations can be made showing the structure of the overlay networks of the Docker Swarm. Services that are running and are listening for connections can also be shown in the visualization.

7 Conclusions

In this research we have explored what can be done on a Docker Swarm overlay network when a host or container gets compromised. Classic layer 2 attacks were tested on the overlay network and experiments conducted show that these attacks do not succeed. We have seen that in certain cases it is possible to inject packets in the VXLAN tunnel, however, setting up a bidirectional connection with one of the VTEPs was not possible. Replaying encrypted traffic from an encrypted Swarm overlay network turned out to be successful. In addition to testing what is possible on a Docker Swarm overlay network strategies of finding what is exposed have been discussed. Finally it was proved to be possible to make visualizations showing the structure of the overlay networks and ports that are in the listening state on the containers. This enables identifying and assessing the security boundaries within the Docker Swarm overlay networks of an organization. The code for the graph visualization is made available on Github at `https://github.com/marcelbrouwers/swarmoverlayvisualizer` [28].

8 Future Work

In future studies it would be interesting to research the (Serf) gossip protocol that is being used for updating the MAC to IP mapping of the VTEP interfaces. The traffic generated by this protocol seems to be encrypted. If it is possible to inject data in the gossip protocol it might become feasible to add a fully functioning interface on the VXLAN on a host that is not part of the Swarm.

References

[1] M. Mahalingam et al. Rfc 7348: Virtual extensible local area network (vxlan): A framework for overlaying virtualized layer 2 networks over layer 3 networks. 2017. Accessed: 2017-06-07 https://tools.ietf.org/html/rfc7348.

[2] Adrian Mouat. 5 security concerns when using docker. 2016. Accessed: 2017-07-01 https://www.oreilly.com/ideas/five-security-concerns-when-using-docker.

[3] Maria Korolov. As containers take off, so do security concerns. 2015. Accessed: 2017-07-01 http://www.csoonline.com/article/2984543 /vulnerabilities/as-containers-take-off-so-do-security-concerns.html.

[4] Docker Project. Docker security. 2017. Accessed: 2017-06-22 https://docs.docker.com/engine/security/security/.

[5] Guido Pineda Reyes. Security assessment on a vxlan-based network. 2014. Accessed: 2017-06-06 http://www.delaat.net/rp/2013-2014/p57/report.pdf.

[6] Siem Hermans and Patrick de Niet. Docker overlay networks. 2016. Accessed: 2017-06-06 https://www.os3.nl/_media/2015-2016/courses/rp1/p50_report.pdf.

[7] Arne Zismer. Performance of docker overlay networks. 2016. Accessed: 2017-06-06 https://esc.fnwi.uva.nl/thesis/centraal/files/f345928229.pdf.

[8] Ramaswamy Chandramouli. Secure virtual network configuration for virtual machine (vm) protection. 2016. Accessed: 2017-06-07 http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-125B.pdf.

[9] Docker swarm mode overlay network security model. 2017. Accessed: 2017-06-07 https://docs.docker.com/engine/userguide/networking/overlay-security-model/.

[10] Raft consensus in swarm mode. 2017. Accessed: 2017-06-23 https://docs.docker.com/engine/swarm/raft/.

[11] Chris Hines. Webinar qa: Docker networking. 2017. Accessed: 2017-07-01 https://blog.docker.com/2016/01/webinar-qa-docker-networking/.

[12] Docker. Libnetwork overlay driver on github. 2017. Accessed: 2017-07-01 https://github.com/docker/libnetwork/blob/master/drivers/overlay/overlay.go Lines: 18, 44, 244.

[13] Mano Marks. Docker swarm visualizer. 2017. Accessed: 2017-06-27 https://github.com/dockersamples/docker-swarm-visualizer.

[14] Erik Lupander. Dvizz - a docker swarm visualizer. 2016. Accessed: 2017-07-02 https://github.com/eriklupander/dvizz.

[15] IANA. Service name and transport protocol port number registry. 2017. Accessed: 2017-07-02 https://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml?search=4789.

[16] Docker Project. Docker swarm mode overlay network security model. 2017. Accessed: 2017-06-26 https://docs.docker.com/engine/userguide/networking/overlay-security-model/.

[17] Chris Hines. Webinar qa: Docker networking. 2016. Accessed: 2017-06-26 https://blog.docker.com/2016/01/webinar-qa-docker-networking/.

[18] Xen Project. Why xen project? 2017. Accessed: 2017-07-08 https://www.xenproject.org/users/why-the-xen-project.html.

[19] Dominic Romeo Sean Whalen, Sophie Engle. An introduction to arp spoofing. 2001. Accessed: 2017-07-08 http://www.leetupload.com/database/Misc/Papers/arp_spoofing_slides.pdf.

[20] Welcome to the ettercap project. 2017. Accessed: 2017-07-08 https://ettercap.github.io/ettercap/.

[21] Dug Song. Dsniff. 2017. Accessed: 2017-07-08 https://www.monkey.org/~dugsong/dsniff/.

[22] Nmap security scanner. 2017. Accessed: 2017-07-08 https://nmap.org/.

[23] Sebastiaan van Stijn. Docker service create doesn't allow –privileged flag. 2017. Accessed: 2017-07-28 https://github.com/moby/moby/issues/24862.

[24] D3 Plus. D3 plus. 2017. Accessed: 2017-07-28 https://d3plus.org/.

[25] Docker project. Docker engine api and sdks. 2017. Accessed: 2017-07-28 https://docs.docker.com/engine/api/.

[26] D3 Plus. Simple static network. 2017. Accessed: 2017-07-28 http://d3plus.org/examples/basic/9042919/.

[27] Mike Bostock. Hierarchical edge bundling. 2017. Accessed: 2017-07-09 https://bl.ocks.org/mbostock/7607999.

[28] Marcel Brouwers. Swarm overlay visualiser. https://github.com/marcelbrouwers/swarmoverlayvisualizer, 2017. Scripts resulting from this project made available in a github repository. Commit: cdf411f5d43186c94a85d30b251b9f4c33b27cb3.

[29] Mike Bostock. Circle packing. 2017. Accessed: 2017-07-09 https://bl.ocks.org/mbostock/4063530.

A Appendix A

```
#!/usr/bin/env python
import nmap
import netifaces as ni
import csv
import socket
import StringIO
import requests
import json
from netaddr import IPAddress

hostname = socket.gethostname()

for interface in ni.interfaces():
    ni.ifaddresses(interface)
    ip = ni.ifaddresses(interface)[2][0]['addr']
    netmask = ni.ifaddresses(interface)[2][0]['netmask']
    cidr = IPAddress(netmask).netmask_bits()
    if ip != '127.0.0.1' and not ip.startswith('172.18'): #limit to the docker swarm
    overlay networks

        nm = nmap.PortScanner()
        network = str(ip) + '/' + str(cidr)
        nm.scan(hosts=network, arguments='')
        reader = csv.reader(StringIO.StringIO(nm.csv()), delimiter=';')
        for line in reader:
            if line[0] != 'host':
                print line[0],line[3],line[4],line[6]
            data = {}
            data['ip'] = line[0]
            data['protocol'] = line[3]
            data['port'] = line[4]
            data['host'] = hostname
            json_data = json.dumps(data, ensure_ascii='False')
            r = requests.post('http://bastia.studlab.os3.nl/rp2/openportreporternmap.
    php', verify=False, json=json_data)
            headers = {'Content-type': 'application/json'}
```

Listing 1: *Python script for scanning subnets of connected networks*

```
import subprocess
import os
import requests
import json
import socket

def get_containers_running():
    return (subprocess.getoutput("docker ps | cut -d \' \' -f 1 | tail -n +2"))

def get_pid_for_container(container):
        return (subprocess.getoutput("docker inspect -f \'{{.State.Pid}}\' %s" % container))

def get_open_ports(pid):
        return (subprocess.getoutput("nsenter -t %s -n netstat -ntlpu | tail -n +3" % pid))

def get_iptables(pid):
    return (subprocess.getoutput("nsenter -t %s -n iptables -L" % pid))

for container in get_containers_running().split(os.linesep):
     print (get_pid_for_container(container))
     netstatoutput = get_open_ports(get_pid_for_container(container)).split("\n")
     for line in netstatoutput:
        line_list = line.split()
        data = {}
        data['host'] = socket.gethostname()
        data['container'] = container
        data['protocol'] = line_list[0]
        data['localaddress'] = line_list[3]
        data['foreignaddress'] = line_list[4]
        data['program'] = line_list[-1]
        json_data = json.dumps(data, ensure_ascii = 'False')
        r = requests.post('http://bastia.studlab.os3.nl/rp2/openportreporter.php', verify=False, json=json_data)
        headers = {'Content-type': 'application/json'}

     iptablesoutput = get_iptables(get_pid_for_container(container)).split(os.linesep)
     print(container)
     if iptablesoutput[0] == "Chain INPUT (policy ACCEPT)":
         policy = "ACCEPT"
     if iptablesoutput[0] == "Chain INPUT (policy DROP)":
         policy = "DROP"
     data = {}
     data['host'] = socket.gethostname()
     data['container'] = container
     data['inputpolicy'] = policy
     print(data)
     json_data = json.dumps(data, ensure_ascii='False')
     r = requests.post('http://bastia.studlab.os3.nl/rp2/firewallreporter.php', verify=False, json=json_data)
     headers = {'Content-type': 'application/json'}
```

Listing 2: *Python script to retrieve firewall status and listening ports for network namespaces that are used by Docker containers. This script is meant run as a cronjob on each of the Docker hosts.*

```
<?php
include_once 'config.php';
$data = json_decode(json_decode(file_get_contents('php://input')), true);

$stmt = $db->prepare("INSERT INTO scan(ip,protocol,port,timestamp,host) VALUES(:ip,:protocol,:port,:timestamp,:host)");
$stmt->execute(array(':ip' => $data['ip'], ':protocol' => $data['protocol'], ':port'
    => $data['port'], ':timestamp' => time(), ':host' => $data['host']));
$affected_rows = $stmt->rowCount();

?>
```

Listing 3: *PHP script for saving the information from the network scanner in a database*

```
CREATE TABLE `firewall` (
  `id` int(11) NOT NULL,
  `host` varchar(255) NOT NULL,
  `container` varchar(255) NOT NULL,
  `inputpolicy` varchar(10) NOT NULL,
  `timestamp` varchar(60) NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
CREATE TABLE `ports` (
  `id` int(11) NOT NULL,
  `host` varchar(255) NOT NULL,
  `container` varchar(100) NOT NULL,
  `protocol` varchar(6) NOT NULL,
  `localaddress` varchar(255) NOT NULL,
  `foreignaddress` varchar(255) NOT NULL,
  `program` varchar(255) NOT NULL,
  `timestamp` int(16) NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
CREATE TABLE `scan` (
  `id` int(11) NOT NULL,
  `ip` varchar(255) NOT NULL,
  `protocol` varchar(5) NOT NULL,
  `port` int(6) NOT NULL,
  `timestamp` varchar(100) NOT NULL,
  `host` varchar(255) NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
ALTER TABLE `firewall`
  ADD PRIMARY KEY (`id`);

ALTER TABLE `ports`
  ADD PRIMARY KEY (`id`);

ALTER TABLE `scan`
  ADD PRIMARY KEY (`id`);

ALTER TABLE `firewall`
  MODIFY `id` int(11) NOT NULL AUTO_INCREMENT, AUTO_INCREMENT=96;

ALTER TABLE `ports`
  MODIFY `id` int(11) NOT NULL AUTO_INCREMENT, AUTO_INCREMENT=28881;

ALTER TABLE `scan`
  MODIFY `id` int(11) NOT NULL AUTO_INCREMENT, AUTO_INCREMENT=2;
```

Listing 4: *Database structure*

```php
<?php
include_once 'config.php';
$data = json_decode(json_decode(file_get_contents('php://input')), true);

$stmt = $db->prepare("INSERT INTO ports(host,container,protocol,localaddress,
    foreignaddress,program,timestamp) VALUES(:host,:container,:protocol,:localaddress
    ,:foreignaddress,:program,:timestamp)");
$stmt->execute(array(':host' => $data['host'], ':container' => $data['container'], ':
    protocol' => $data['protocol'], ':localaddress' => $data['localaddress'], ':
    foreignaddress' => $data['foreignaddress'], ':program' => $data['program'], ':
    timestamp' => time()));
$affected_rows = $stmt->rowCount();

$stmt = $db->prepare("DELETE FROM ports WHERE container = ? AND timestamp <= ?");
$time = time() - 60;
$stmt->execute(array($data['container'], $time));
?>
```

Listing 5: *PHP script for saving the listening ports information from the Python script in listing 2 in a database*

```php
<?php
include_once 'config.php';
$data = json_decode(json_decode(file_get_contents('php://input')), true);

$stmt = $db->prepare("INSERT INTO firewall(host,container,inputpolicy,timestamp)
    VALUES(:host,:container,:inputpolicy,:timestamp)");
$stmt->execute(array(':host' => $data['host'], ':container' => $data['container'], ':
    inputpolicy' => $data['inputpolicy'], ':timestamp' => time()));
$affected_rows = $stmt->rowCount();

$stmt = $db->prepare("DELETE FROM firewall WHERE container = ? AND timestamp <= ?");
$time = time() - 60;
$stmt->execute(array($data['container'], $time));
?>
```

Listing 6: *PHP script for saving the firewall status information from the Python script in listing 2 in a database*

```php
<?php
include_once 'config.php';
$tasks = json_decode(file_get_contents('http://'.$dockerapihost.'/tasks'));
$engines = json_decode(file_get_contents('http://'.$dockerapihost.'/nodes'));
$networks = json_decode(file_get_contents('http://'.$dockerapihost.'/networks'));
$services = json_decode(file_get_contents('http://'.$dockerapihost.'/services'));

$nodes = array();
$connections = array();
foreach ($networks as $network) {
    if($network->{'Name'}!="bridge" and $network->{'Name'}!="host" and $network->{'
    Name'}!="docker_gwbridge" and $network->{'Name'}!="none"){
    $arradd['id'] = $network->{'Id'};
    $arradd['group'] = $network->{'Id'};
    $arradd['name'] = $network->{'Name'};
    $arradd['size'] = 10;
    if($network->{'Attachable'}=="true"){$attachable="yes";}
    else{$attachable="no";}
    $subnet = json_decode(file_get_contents('http://'.$dockerapihost.'/networks/'.
    $network->{'Id'}), true)['IPAM']['Config'][0]['Subnet'];

    $arradd['tooltip'] = "Name: " . $network->{'Name'} . "<br />VNI: " . $network->{'
    Options'}->{'com.docker.network.driver.overlay.vxlanid_list'} . "<br />Attachable
    : " . $attachable . "<br />Subnet: " . $subnet;
    $arradd['firewall'] = "";
    $connection['source'] = $network->{'Id'};
    $connection['target'] = $network->{'Id'};
    array_push($nodes, $arradd);
    }

}

foreach ($tasks as $key => $task) {
  if($task->{'Status'}->{'State'}=="running"){

    $arradd['id'] = $task->{'Status'}->{'ContainerStatus'}->{'ContainerID'};
    $arradd['group'] = "node";
    $arradd['name'] = substr($task->{'Status'}->{'ContainerStatus'}->{'ContainerID'},
     0, 12);
    $arradd['size'] = 5;

    $containerid = $arradd['name'];
    $image = substr($task->{'Spec'}->{'ContainerSpec'}->{'Image'}, 0, 15);
    $pid = $task->{'Status'}->{'ContainerStatus'}->{'PID'};
    $arradd['firewall'] = "";
    foreach($db->query("SELECT * FROM firewall WHERE container='$containerid'") as
    $row) {
      $arradd['firewall'] = $row['inputpolicy'];
    }
    foreach ($services as $service){
      if($task->{'ServiceID'} == $service->{'ID'}){$servicename = $service->{'Spec
    '}->{'Name'};}
    }
    foreach($engines as $node){
      if($task->{'NodeID'} == $node->{'ID'}){
        $residesonnode = $node->{'Description'}->{'Hostname'};
      }
    }


    $arradd['tooltip'] = "ID: " . $containerid . "<br/>Image: " . $image . "<br />PID
    : " . $pid . "<br />FW Input Chain: " . $arradd['firewall'] . "<br />Servicename:
     " . $servicename . "<br />Node: " . $residesonnode;
    array_push($nodes, $arradd);


    foreach($db->query("SELECT * FROM ports WHERE container='$containerid'") as $row)
     {
      $arradd['id'] = $task->{'ID'}.'.'.$row['localaddress'];
      $arradd['group'] = "port";
      $arradd['name'] = $row['localaddress'];
      $arradd['size'] = 3;
      $arradd['tooltip'] = "Program: " . $row['program'] . "<br />Protocol: " . $row['protocol'] . "<br />Local address: " . $row['localaddress'] . "<br />Foreign Address: " . $row['foreignaddress'];

      $connection['source'] = $task->{'Status'}->{'ContainerStatus'}->{'ContainerID'};
      $connection['target'] = $task->{'ID'}.'.'.$row['localaddress'];
      if(strpos($row['localaddress'], '127.0.0') === false){
      array_push($nodes, $arradd);
      array_push($connections, $connection);
      }

    }

    foreach ($task->{'NetworksAttachments'} as $networkattachment => $network) {
      $connection['source'] = $task->{'Status'}->{'ContainerStatus'}->{'ContainerID'};
      $connection['target'] = $network->{'Network'}->{'ID'};
      array_push($connections, $connection);
    }
  }
}

$output = array();
$output['nodes'] = $nodes;
$output['links'] = $connections;
print json_encode($output, JSON_PRETTY_PRINT);
?>
```

Listing 7: *PHP script for generating the data used for the visualization with the graph showing overlay networks and listening ports*

```
<?php
include_once 'config.php';
?>
<!DOCTYPE html>
<meta charset="utf-8">
<style>

.links line {
  stroke: #000000;
  stroke-opacity: 1;
}

.nodes circle {
  stroke-width: 2.5px;
}
.text {
  font: 12px helvetica;
  color: rgba(0, 0, 0, 0.3);
  pointer-events: none;
}
```

```
div.tooltip {
  position: absolute;
  text-align: left;
  width: 200px;
  height: 150px;
  padding: 2px;
  font: 14px sans-serif;
  background: #aaaaaa;
  border: 0px;
  border-radius: 8px;
  pointer-events: none;
}

</style>
<svg width="1400" height="1000"></svg>
<script src="https://d3js.org/d3.v4.min.js"></script>
<script>

var svg = d3.select("svg"),
    width = +svg.attr("width"),
    height = +svg.attr("height");

var color = d3.scaleOrdinal(d3.schemeCategory20);

var simulation = d3.forceSimulation()
    .force("link", d3.forceLink().id(function(d) { return d.id; }).distance(100).
    strength(0.6))
    .force("charge", d3.forceManyBody().strength(-100))
    .force("center", d3.forceCenter(width / 2, height / 2));



d3.json("portnetworkdata.php", function(error, graph) {
  if (error) throw error;

  var link = svg.append("g")
      .attr("class", "links")
    .selectAll("line")
    .data(graph.links)
    .enter().append("line")
      .attr("stroke-width", 2);
var div = d3.select("body").append("div")
    .attr("class", "tooltip")
    .style("opacity", 0);

  var node = svg.append("g")
      .attr("class", "nodes")
    .selectAll("circle")
    .data(graph.nodes)
    .enter().append("circle")
      .attr("r", function(d) { return d.size*2; })
      .attr("fill", function(d) { return color(d.group); })
      .attr("stroke", function(d){ if(d.firewall === 'ACCEPT'){return '#FF0000';}else
     if(d.firewall === 'DROP'){return '#00FF00';}else{return '#000000';}})
      .call(d3.drag()
          .on("start", dragstarted)
          .on("drag", dragged)
          .on("end", dragended))
          .on("mouseover", function(d) {
            div.transition()
                .duration(200)
                .style("opacity", .9);
            div .html(d.tooltip)
                .style("left", (d3.event.pageX) + "px")
                .style("top", (d3.event.pageY - 28) + "px");
            })
        .on("mouseout", function(d) {
            div.transition()
                .duration(500)
                .style("opacity", 0);
        });

  var text = svg.append("g")
  .attr("class", "text")
    .selectAll("text")
    .data(graph.nodes)
    .enter().append("text")
      .text(function(d) { return d.name; })
        .call(d3.drag()
          .on("start", dragstarted)
          .on("drag", dragged)
          .on("end", dragended));



  simulation
      .nodes(graph.nodes)
      .on("tick", ticked);

  simulation.force("link")
      .links(graph.links);

  function ticked() {
    link
        .attr("x1", function(d) { return d.source.x; })
        .attr("y1", function(d) { return d.source.y; })
        .attr("x2", function(d) { return d.target.x; })
        .attr("y2", function(d) { return d.target.y; });

    node
        .attr("cx", function(d) { return d.x; })
        .attr("cy", function(d) { return d.y; });
    text
        .attr("x", function(d) { return d.x; })
        .attr("y", function(d) { return d.y; });
  }
});

function dragstarted(d) {
  if (!d3.event.active) simulation.alphaTarget(0.1).restart();
  d.fx = d.x;
  d.fy = d.y;
}

```

```
function dragged(d) {
  d.fx = d3.event.x;
  d.fy = d3.event.y;
}

function dragended(d) {
  if (!d3.event.active) simulation.alphaTarget(0);
  d.fx = null;
  d.fy = null;
}


</script>
```

Listing 8: *Script for generating the visualization shown in figure 4. The script used the data generated with the script in listing 6*

```
<?php
$tasks = json_decode(file_get_contents('http://145.100.110.50:4243/tasks'), true);
$networks = json_decode(file_get_contents('http://145.100.110.50:4243/networks'),
    true);
$output = array();

foreach ($networks as $overlay){
  if($overlay['Driver']=='overlay'){
  $arrentry['name'] = $overlay['Id'] . '.' . $overlay['Name'];
  $arrentry['size'] = 4000;
  $arrentry['imports'] = '';
  array_push($output, $arrentry);
  }
}

foreach ($tasks as $taskdata) {
  if ($taskdata['Status']['State'] == "running") {

      $arrentry['name'] = $taskdata['ID'];
      $arrentry['size'] = 1000;
      $attachments = array();
      foreach ($taskdata['NetworksAttachments'] as $network) {
        foreach ($networks as $overlay){
          if($overlay['Id']==$network['Network']['ID']){
            $attachment = $network['Network']['ID'] . '.' . $overlay['Name'];
            array_push($attachments, $attachment);
          }
        }
      }
      $arrentry['imports'] = $attachments;
      array_push($output, $arrentry);

  }}

print json_encode(array_values($output), JSON_PRETTY_PRINT);
?>
```

Listing 9: *PHP script for generating the data used for generating figure 3*

```
<?php

$data = json_decode(file_get_contents('http://145.100.110.50:4243/nodes'));
$tasks = json_decode(file_get_contents('http://145.100.110.50:4243/tasks'));
$services = json_decode(file_get_contents('http://145.100.110.50:4243/services'));
//var_dump($data);
$array = array();
foreach ($data as $key => $value) {
  //print $value->{'Description'}->{'Hostname'};
  $arradd['name'] = $value->{'Description'}->{'Hostname'};
  //$arradd['size'] = 200;

  $tasksonnode = array();
  foreach ($tasks as $task => $taskdetail)
  {
    if($taskdetail->{'Status'}->{'State'}=="running"){
    if($taskdetail->{'NodeID'} == $value->{'ID'}){
      $taskfornode['name'] = substr($taskdetail->{'ID'}, 0, 12);
      foreach ($services as $service){
      if($taskdetail->{'ServiceID'} == $service->{'ID'}){$taskfornode['name'] =
    $service->{'Spec'}->{'Name'};}
    }
      $taskfornode['size'] = 50;
      array_push($tasksonnode, $taskfornode);
    }
    }
  }
  //print_r($tasksonnode);
  $arradd['children'] = $tasksonnode;
  array_push($array, $arradd);

}
//print_r($array);

echo "{
 \"name\": \"swarm\",
 \"children\": [";
print substr(json_encode(array_values($array), JSON_PRETTY_PRINT), 1, -1);
echo "]}";

?>
```

Listing 10: *PHP script for generating the data used for generating figure 5*