TECHNISCHE UNIVERSITÄT DRESDEN

Faculty of Computer Science Institute of Systems Architecture, Chair of Computer Networks

Tencrypt: Hardening OpenShift by Encrypting Tenant Traffic

Forschungsprojekt Anwendung

Dominik Pataky

2018-10-31

Abstract

OpenShift is a Kubernetes distribution maintained by Red Hat. It allows for simple setup and orchestration of container-based applications. The main focus of OpenShift lies on an efficient deployment process as well as infrastructure scalability. Even though security is an important cross-section topic, hardening of OpenShift is not a core concept yet.

The aim of this work is the research of possibilities which allow automatic and transparent encryption of internal network traffic between applications of Tenants in a multi-tenant OpenShift infrastructure.

Key feature is the earliest-possible encryption of network packets after packet creation, with a low impact on performance. The usage of dedicated network namespaces in container environments is taken into account. This work discusses different design alternatives. After a well-grounded choice of one design, this approach is evaluated in regards to performance using a prototypic implementation.

Author	Dominik Pataky <dominik.pataky@tu-dresden.de>
	PGP 0x80BF7C9C5B62468F
Supervisor	Dr.-Ing. Marius Feldmann
Last update	Wednesday 14th November, 2018 14:59
License	CC BY-SA 4.0

Contents

1. Introduction

Virtualisation is an important feature in modern computing infrastructure. Be it fully virtualised components known as virtual machines (VMs) or the usage of containerised applications, virtualisation allows more flexible resource usage and deployment than the plain installation of software on bare metal.

But security and data privacy are becoming increasingly bigger factors in the decision, which technologies companies and other user groups take into consideration when migrating software stacks into cloud environments. OpenShift is one of the main platforms for containerised application deployment. Yet, regarding security not much has been done to give guarantees to users of the platform.

In this project, the requirements for a transparent encryption of network traffic between multiple applications (Pods) of a Project inside OpenShift are evaluated and implemented as a proof of concept. Based on these results, measurements are taken, to compare the different throughput performance metrics in both the vanilla and the patched environments. Lastly, the conclusion wraps up the approach, the results and further work to be done.

2. Red Hat OpenShift, Kubernetes and Docker

OpenShift comes in multiple variances: Origin Kubernetes Distribution (OKD), also known as Origin, being the upstream open-sourced platform extending a distribution of Kubernetes with multiple DevOps tools for better application development, deployment and management. OKD also adds more security features to Kubernetes.

To summarise the features of OpenShift, [Hat18] gives a detailed overview. As mentioned above, all hardware or virtual machines in an OpenShift infrastructure are called **Nodes** and are deployed with Red Hat Enterprise Linux (RHEL). The **Master** provides API, Authentication mechanisms, data storage, orchestration, scheduling and health/scaling algorithms.

On the Nodes, **Pods** are deployed to group Containers. Pods belong to Users and Projects. Pod deployment can be controlled with Policies.

The Master/Node network is extended by a **Registry**, the **Persistent Storage**, a **Service Layer** (Service Discovery) and a **Routing Layer**. Access to the infrastructure is possible via web, CLI, IDE adapters and APIs. This access is used for e.g. CI/CD and DevOps. The auto-healing algorithms restart failed Pods and re-deploy Pods from failed Nodes on other Nodes. **Services** in the Service Layer abstract and load-balance services available in Pods, using a fixed IP and key-value identifiers. **Routes** in the Routing Layer expose Services externally, while internal traffic stays in the Service Layer.

Routing uses pluggable components like HAproxy and F5. Supported protocols include HTTP/ HTTPS, WebSockets and TLS with SNI. Non-standard ports may be used with Cloud load-balancers, external IPs and the NodePort policy. Routes can also split traffic for A/B testing, Blue/Green[1] and Canary[2] Deployments. The NodePort policy binds Services to a fixed, unique port on all Nodes inside the network. Nodes which do not host the bound Service/Pod redirect the traffic to a hosting Node. Services can also be bound to an external IP with port, routed dynamically internally. Automatic IP allocation from an Ingress IP pool can be used, as well as IP failover for High Availability (HA). Outgoing traffic can be controlled by an Egress Router.

Networking uses internal DNS servers to address Services by name. Split DNS can be done with SkyDNS, the Master replies to internal queries and other nameservers reply to external requests. Software Defined Networking for unified cluster network and pod-to-pod traffic. OpenShift uses the Kubernetes Container Networking Interface (CNI) (plug-in interface for network-

[1] `https://martinfowler.com/bliki/BlueGreenDeployment.html`
[2] `https://martinfowler.com/bliki/CanaryRelease.html`

ing in Containers) with network plugins: OpenShift Plugin, Flannel, Nuage, Calico, Contrail, Cisco Contiv, Big Switch and VMware NSX-T.

OpenShift SDN has multiple variants: flat network, multi-Tenant network and network policy (see section 2.1). Pod-to-Pod networking is done via VXLAN overlay networks with **Open vSwitch (OVS)**. The OVS packet flow depends on the Pod traffic destination: Container to container on the same host is routed through the virtual bridge, C2C on different hosts is passed through the VXLAN adapter and the usage of Flannel routes according to the Flannel Routing Table.

The logging stack is called EFK for ElasticSearch, Fluentd and Kibana. Access control can be defined for different log types. For example, admins should be able to view all logs, but devs need only access to their container logs. Metrics are collected with Kubernetes Metrics Server (formerly Heapster), Hawkular and Cassandra. Stored metrics can be requested via API (by e.g. web interfaces).

Security takes multiple aspects into account: Container Host & Multi-tenancy, Federated Clusters, Container Platform, API Management, Network Isolation, Deploying Container, Container Registry, Container Content, Storage and Building Containers.

Secret Management is managed by a secured store on the Master. Secrets (credentials, SSH keys, certs, ...) are made available inside Containers through encrypted transit and via environment variables, volume mounts or via external systems. **Storage** objects (Persistent Storages (PVs)) are defined for pieces of network storage (NFS, OpenStack Cinder, iSCSI, Azure Disk, AWS EBS, FlexVolume, GlusterFS, Ceph RBD, Fiber Channel, Azure File, GCE Persistent Disk, VMWare vSphere VMDK). Pods which need storage can request PVs (Claim) from a pool of PVs based on size, access mode, labels and type. Volume provisioning for Pods can also be based on StorageClass and Claims (e.g. a Pod requests the „fastest" storage and gets SSD storage assigned). The result gets passed to the corresponding Provisioner. Red Hat themselves also offer a type of natively integrated storage called Gluster Storage. It is a containerised storage with native integration with OpenShift.

To solve the slowness of manually managed **Service Brokers**, the Open Service Broker API is developed as a multi-vendor project. It aims to standardise how services can be consumed on Cloud-native platforms across providers. Participating bodies include Fujitsu, Pivotal, IBM, Red Hat, Google and SAP.

2.1. Networking concepts

Kubernetes by default allocates IP addresses from private internal network ranges for each Pod. Additionally, Pods receive their own unique networking namespace as isolation. See figure 1 for details. As described in [Hau18], Kubernetes uses the same network namespace for all containers inside a Pod for intra-Pod traffic.

As mentioned above, OpenShift differentiates between three modes of networking when using the SDN overlay network (using VXLAN, [OKDb]):

Flat Using the `ovs-subnet` plug-in, a „flat" Pod network is established. Every Pod can communicate with every other Pod.

Multi-Tenant The `ovs-multitenant` plug-in isolates traffic on the Project-level. Each Project receives a unique Virtual Network ID (VNID) (comparable to a VLAN tag) which restricts traffic flows from pods to pods of the same Project. The special `VNID 0` can be used to enable unrestricted traffic flows, e.g. for load balancers. It can be seen as Project-level network isolation, supporting multicast and egress network policies.

Network Policy Lastly, the `ovs-networkpolicy` plug-in allows the configuration of custom isolation policies. This granular policy-based isolation provides rules like „allow all traffic inside the project" or „allow traffic to project Purple on port 8080".

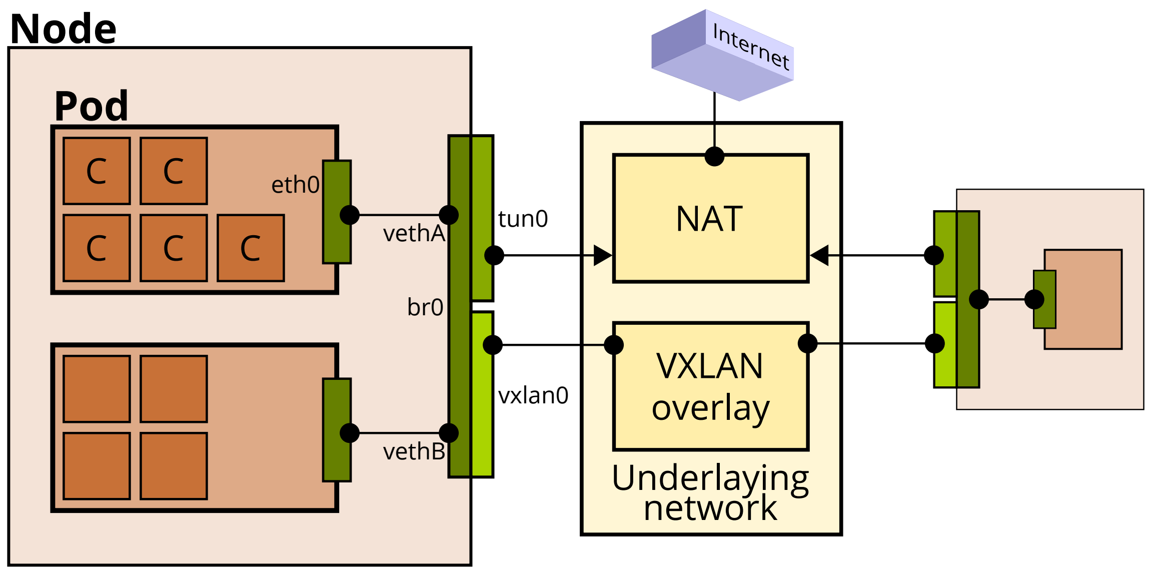

Figure 1: Networking of Nodes, Pods and Containers

Containers are created inside Pods. Containers receive an isolated `eth0` network adapter from a `veth` pair. The other end is added to the OVS bridge `veth0`. More information about this mechanisms in Linux in [Blo].

2.2. Existing container security projects and integrations

OKD Secured Routes When using TLS for traffic coming into an OKD Router, the Router decides how to handle this „secured route“. The „Secured Routes“ feature can then be configured in different variants to handle the corresponding route further continuing in the cluster. It can either be Edge Termination (serving certificates from the Router), Passthrough Termination (TLS is handled by the Service) and Re-encryption Termination (the router provides certificates for the external client, but also encrypts proxied traffic to the Service). If a Route is marked as secure, but no Secured Route is configured, internal traffic is routed unencrypted.

IPsec The OKD project docs reference IPsec to be used to encrypt traffic between Master and Nodes, and for Node-to-Node communications. IPsec is an IETF standard, published as RFCs 4301 and others.

Aporeto Aporeto provides a set of security features usable with Kubernetes in different Cloud infrastructures, including OpenShift. It implements End2End container encryption with TLS. They also develop „Aporeto integration with Kubernetes Network Policies“, which integrate into the Kubernetes policy catalogues.

Aqua Security Container Security for Red Hat OpenShift. Enables container-level firewalls, but no encryption techniques.

Istio Auth Istio uses Envoy service proxies inside Pods. These Envoys can be used for mTLS-encrypted Pod-to-Pod (Service-to-Service) traffic. Uses the Secure Production Identity Framework for Everyone (SPIFFE) authentication framework.

It is also worth taking a look into the OpenShift Security roadmap [sec-road], which keeps track of the implemented and proposed security features by OpenShift version.

2.3. Related technologies

During the project run, related technologies in the context of virtualisation, SDN and encryption were looked into. The following list highlights the most interesting ones, which might also be used (or have to be taken into consideration) in further development.

Single Root I/O Virtualization (SR-IOV) With SR-IOV, PCI Express hardware resources provide an additional layer for virtualisation use cases. Hardware with SR-IOV support offer one so called Physical Function (PF, full access to the resource) and a configurable amount of Virtual Functions (VF, only supporting IO operations). This way, virtual machines (VMs) can share a physical device multiple times, while each VM sees the shared resource as an exclusive part of the virtualised system. [Low09]

MACsec MACsec provides multiple security enhancements, noticeably encryption and integrity on layer 2. It extends Ethernet frames by adding a new EtherType and a MACsec tag, inserted after the source MAC address. MACsec is standardised as IEEE 802.1AE.

Cisco Application Centric Infrastructure (ACI) ACI is an SDN infrastructure oriented, policy-based framework developed by Cisco. It aims to be used for better management and uses common technologies like VXLAN, Equal-cost Multipath (ECMP) routing and SDN controllers.

3. Security requirements and threat model

Using complex software infrastructures for computation and storage introduces multiple attack vectors. To analyse the security requirements Tencrypt should cover, the STRIDE/AINCAA list given in [Sho14] is taken into consideration. STRIDE can be viewed as a super-set of the CIA triad.

Authentication Pods which receive traffic over an intra-Project connection from other Pods of the same Project must be able to ensure the authenticity of the traffic sender.

Integrity Data transmitted over a Tencrypt channel must support validation of integrity.

Non-repudiation *Not a subject of this project.*

Confidentiality The encryption of data sent to other Pods must not be readable by outsiders, e.g. administrators listening on the `br0` bridge. *Introduced in this paper.*

Availability is one of the key elements of the underlying Kubernetes orchestration tool. Kubernetes offers features like load balancing, migration of Pods in case of failures, automatic scaling of resources in case of increased load and e.g. dynamic routing, with which any changes in the container topology can be transparently handled without interruptions. *An extended implementation of Tencrypt should handle discovery of new routes and other interference.*

Authorisation Tencrypt-enabled Pods should automatically reject unencrypted or malformed traffic from Project-internal Pods, reducing the threat of forged traffic from an attacker masking as a Project member.

A note regarding Containerisation

A note before we go through the list of possible attack vectors and threats given in the OpenShift Pod setup: this threat analysis does not include the threat of a hostile administrator or attacker controlling the hosted Node (and especially the network bridge). The isolation of Pods in namespaces does not help in any way to e.g. prevent traffic interception by a third party, if this third party has unlimited access on the host machine. To prevent administrative access into the underlying network, mount and PID namespaces, an entirely different approach is needed. Tencrypt focuses on the transparent encryption of Project-internal Pod-to-Pod traffic, on the same Node and in cross-Node traffic.

Nevertheless, the results presented in this paper should be seen in context with currently expanding technologies which support hardware-based isolation of Containers. Once the isolation of namespaces can be technically ensured, the `eth0` network interfaced used by Tencrypt would be placed at the boundary between the protected container and the shared resources on the host.

Additionally, the reduction of possible attack points on hosts and inside the network might help mitigating threats in an administration with strict hierarchical access roles. If, for example, a group of administrators only has debug access to the networking bridge, but not to the underlying host system, Tencrypt would prevent disclosure of intra-Project data exchanges to members of this administrative group.

Threats

The following table in figure 2 gives an overview over possible threats which Tencrypt takes into consideration. More information about security in OpenShift can be found in [OKDa]. For details on Pod permissions in OpenShift, the OKD Security Context Constraints [OKDc] chapter extends this section.

ID	Description	Mitigation
T1	An attacker uses a Pod to intercept traffic originating from other namespaces (Pods) on the `br0` bridge.	Encryption of traffic, hardening of isolation mechanisms (Linux kernel).
T2	An attacker not only intercepts, but is able to modify traffic on the `br0` bridge or the `vxlan0` adapter.	Encryption and integrity checks.
T3	Interception and modification of Master-to-Node traffic.	IPsec, as mentioned in section 2.2.
T4	Interception of Node-to-Node traffic, both Project-internal and cross-Project.	A combination of Node-to-Node IPsec and Tencrypt for Project-internal traffic
T5	Incoming external Service traffic is intercepted (and maybe modified) before it reaches the handling Service namespace.	Secured routes as mentioned in section 2.2.
T6	The Pod image used by OpenShift to deploy new Pods, is maliciously modified.	Securing the image registry. *The registry depends on the used container technology and might be an external component.*
T7	Resources requested by a Pod limit the availability of other Pods on the same Node.	Continuous resource monitoring, migration or halting of resource intensive Pods if needed.

Figure 2: Table of possible threats related to the network stack

4. Encrypting traffic between Pods

4.1. Fundamentals

To find a possible solution for most-early integration of transparent Tenant-level encryption, the following basic policies were defined:

1. When speaking of „encrypted traffic“, it only includes Tenant-internal (Project-internal) traffic, not egress traffic leaving the platform or traffic exchanged with Services of other Projects.
2. Containers inside Pods in OpenShift share a virtual ethernet interface, `eth0`. This interface will be the main implementation focus. All packets transmitted over the `veth` pair via the `br0` bridge and the `vxlan0` interface (see figure 1) towards other Pods of the same Project should be encrypted.
3. For this experiment, only data of OSI layer 5-7 is encrypted, also known as application layer in the TCP/IP stack. This reduces the overhead which would be introduced by layer 3 or even layer 2 encryption, most probably resulting in encapsulation and additional masking techniques. *The proof of concept implementation does use UDP encapsulation, because tunneling TCP with only payload encryption proved to be not feasible.*

4.2. Ideas and possible approaches

After the analysis of the OpenShift network architecture, combined with the security requirements, the following design and implementation approaches were collected and served as the base for further implementation.

1. Using simple AES, a shared Secret can be used as the encryption and decryption key. Payload of packets would need to be transparently encrypted using the key and decrypted on arrival.
   - How to handle payload size growth and MTU?
   - How can the keys be shared and rotated?
   - Does this approach fulfil the security requirements?
2. In contrast to symmetric encryption with AES, asymmetric encryption with periodically changed session keys could be used (hybrid encryption). With this approach, Pods could put their public key into the Secret storage, not having to share the private key.
   - Who generates the key pair? (the Pod instance?)
   - Which established system could be used? (e.g. X.509)
   - Is this approach realisable without introducing a new component into OpenShift (as would be needed for certificate chains)?
3. Wireguard is one possible existing tunnel encryption software and features moving created endpoints into network namespaces. Could Wireguard therefore be used to deploy network interfaces inside containers?
   - Does it scale?
   - Can the compiled Wireguard tools and kernel modules be integrated at all?
   - Which component would create the Wireguard interfaces?
   - Can the Secret storage be used to receive all existing public keys from peers?

4.3. Using Minishift for experimental implementations

With Minishift developers can locally deploy a virtual machine (VM) with a Dockerised OpenShift cluster within it. It is part of the official OKD project and one possible way of setting up a development environment which is close to production environments in deployed OpenShift clusters. The alternative is the deployment of native Docker containers on the development machine. Earlier (according to a blog post from 2015), a Vagrant VirtualBox image was provided for this case, but it seems to be no longer maintained.

Minishift, as a fork from the Kubernetes Minikube project, uses Docker Machine for deployment. In the Tencrypt setup, the Virtualbox VM driver was used, because it runs out-of-the-box. In contrast, the KVM setup needs additional tools from the Docker Machine project. Running the VM, at first the Boot2Docker VM image was deployed, but was later replaced by the CentOS image. See section 4.4.1 for details.

The local „developer“ account was used for administrating deployments. For testing purposes, two Projects were created, with the first having two Deployments (two running Pods) while the second had one. With this environment, inter-Project reachability and later intra-Project traffic encryption can be tested.

Minishift works with the default `ovs-subnet` method, permitting traffic from all Pods to all other Pods, regardless of the Project. To further test the functionalities of OpenShift, the `ovs-multitenant` network policy should be used, isolating Pods of specified Projects. Yet, configuring it did not result in expected policy changes. An issue on Github states, that the plugin currently does not work with Minishift, as the feature is not implemented.[3]

4.3.1. Dissecting the network configuration in Minishift

When using Minishift, the OpenShift cluster, which one would distribute over multiple hardware nodes (separating the Master, etcd and the compute Nodes), is simulated with Docker containers. The localhost (the VM) is pre-configured as the Node on which Pods are deployed. Containers inside these Pods are ran as Docker containers bound to the same Docker daemon as the Pod instance. Inspecting the Minishift networking shows that the VM uses three interfaces, `eth0`, `eth1` and `docker0`.

The interface `eth0` with the range `10.0.2.15/24` is Virtualbox and KVM specific and is used for host-communication via SSH.The `eth1` interface is configured as the `vboxnet0` interface on the original host, and uses an address range like `192.168.99.100/24`.

The `docker0` bridge uses `172.17.0.1/16` for all Pod namespaces. Each Pod is configured with an IP from this range in its isolated namespace, connected to the bridge via `veth` pairs. Configured Services receive a „Cluster IP“ in the network range `172.30.0.0`. These Virtual IPs (VIPs) are routed with `iptables` NAT rules on the Minishift host. If a service is requested by its DNS name, OpenShift resolves it to the `172.30.x.x` address.

4.3.2. Secret management to share pre-shared keys

To test the possibility of using pre-shared keys within the Pods and Containers, the integrated Secret management component of OpenShift was evaluated. With it, one enters a Project, creates a Secret „Tencrypt“ resource and give it the key „`PROJECT_PSK`“ with a random value. There are two ways to access this key from within a Pod:

Environment variable The key/value entry is accessible via the generated environment variables inside the container. E.g. a script could use `$TENCRYPT_PROJECT_PSK` if configured with this key.

[3] `https://github.com/minishift/minishift/issues/1167`, last visit on 2018-10-03

Filesystem mount The Secret with all its keys is mounted as a volume and can be accessed by reading the mounted files. E.g. the Container can execute `cat /mnt/tencrypt/project_psk` when mounted under `/mnt/tencrypt`.

Further looking into the availability of `ENV` and mounts in the Pod container revealed, that each container has its own environment. The Pod container does not receive the Secret environment entries, if configured. Testing the mount namespaces for the second way of sharing Secrets showed that file system mounts are also not shared between the Pod and its application containers. The second option was therefore not applicable as well. This leads to the conclusion that the existing Secret Storage might not be sufficient for this type of Secret sharing.

4.3.3. Connecting to the Docker daemon and building images

As mentioned, inside the VM a Docker daemon handles the building and orchestration of images. The internal registry contains some images used to deploy OpenShift components, e.g. the `openshift/origin-haproxy-router` and more interestingly `openshift/origin-pod`. Minishift allows to connect to this Docker daemon instance by executing `eval $(minishift docker-env)` and then further using the local `docker` tool which connects to the remote daemon.

4.3.4. Patching the Pod image

The connection to the integrated Docker daemon as explained above also gives access to the VM-internal Docker registry with the mentioned `openshift/origin-pod` image. Using a `Dockerfile` and some Docker commands, we can re-tag the original Pod image as shown in listing 5. The Dockerfile for building the new image is located in `../implementations/origin-pod/`. The list of images is shortened, the full list includes other component images as well.

The new image is loaded when OpenShift deploys a new Pod (in the default configuration, which uses the identifier `openshift/origin-{component}:v3.10` as image name). This can be verified by using the developer web interface or the `oc` CLI tool to re-deploy an application (in this example, the three applications nginx-ex, django-ex and cakephp-ex).

As soon as the application is re-deployed, both, a new Pod container and a new application container, are visible in the Docker container list. Using `docker ps -n2`, the two most recently created containers are displayed, in which a shell can be openend by using `docker exec -ti -u root <id> bash`. The verification is complete when `ls /etc/hello_test` executes successfully. We now have a new layer on top of the default `origin-pod` image.

4.4. Implementation concepts

As seen above in figure 1, Pods receive one side of a virtual ethernet (`veth`) network adapter pair, connected to `br0`. Inside the Pod, all Containers share this network interface as `eth0`. Technically, the whole package is a collection of multiple containers, sharing the same network namespace. The Pod container is special, since it only executes the `/usr/bin/pod` binary, which only waits for an interrupt and does nothing else, keeping the container alive as long as the namespace resources are needed for the containers belonging to this Pod.

Since the encryption should be done at the earliest possible point in the network stack, the modification of the `eth0` adapter is the primary target. Secondly, it should not be necessary to patch any deployed application image used in OpenShift. Developers and users of the platform should not need to take any measures regarding their deployments if they wish to use Tencrypt.

In the following, part 1 shows how to implement a transparent proxy for Service to Service connections inside the Pods and their Containers. This is followed by part 2, in which the differentiation between Project-internal and -external traffic is examined. In part 3 possibilities for encryption and decryption are evaluated, leading to the proof of concept implementation in section 4.5.

Figure 3 shows the implementation schema with the proxy application which reads packets from the virtual TUN interface `tenc0`. Containers route their packets through this interface, as it is configured as the default gateway for hosts in the Services IP address range. The proxy application itself uses the interface address from `eth0` to relay the packets over `eth0`.

4.4.1. Part 1: Setting up the Pods network

This part started with a simple step in which the possibilities of network interface, address and route manipulation from inside the container were tested. After having the possibility to add binaries and scripts into the Pod image, the Dockerfile copied a shell script which was executed on Pod start. The script contained some basic commands like `ip addr add 172.17.0.99/16 dev eth0` (background: the first attempt aimed at creating additional IP addresses which could be addressed Project-internally). Adding a simple Go binary which read all available network interface addresses worked as well.

Trying to configure any routes or addresses on the `eth0` interface fails however. After some research and a look at the Docker capabilities setup, the problem could be identified to be a missing capability: `NET_ADMIN`. Without it, even a root inside the container may not manipulate the network interface.

To verify that this capability is indeed the missing piece, a simple test is sufficient: running a Docker container based on the patched Pod image with the `--cap-add NET_ADMIN` flag creates a new Pod container in the background. Attaching to it with a console and running `ip addr add 127.0.0.2/8 dev lo` adds a new IP address to the `lo` interface without problems.

Finding a solution for this problem included multiple approaches:

1. An extensive search in the Origin and Kubernetes code repository, to investigate if the Pod container setup could be patched to grant additional capabilities. This brought no results, the Pod setup is complex, abstracted on multiple levels inside OpenShift and the Kubernetes libraries.
2. Extending the Dockerfile with `setcap` and `iptables` commands. Result: `setcap` works, but does nothing later when executed inside the container, `iptables` fails because of the known capability restrictions.
3. Adding a new Security Context Constraint (SCC) with the capability and adding `securityContext.capabilities.add: ["NET_ADMIN"]` to the deployment config. No improvements.
4. Updating the `hostConfig` of a running Docker container, adding `NET_ADMIN` to `CapAdd` and restarting the container. Tried both, the manipulation of the file on the host system and any possibilities offered by the Docker Python library. This did not work as expected [4] [5].
5. Changing capabilities as root from the host machine, manipulating the aufs file system of the container in `/var/lib/docker/aufs`. This also failed, because `setcap` did not work because of missing symbols on TinyCoreLinux (which is used in Boot2Docker), even when self-compiled.

At this point it was obvious that too much time went into investigating a problem which needed to be solved, but would probably not take this much effort later, when Tencrypt project

[4] `https://serverfault.com/questions/861227`, last visit on 2018-09-20
[5] `https://stackoverflow.com/questions/38758627`, last visit on 2018-09-20

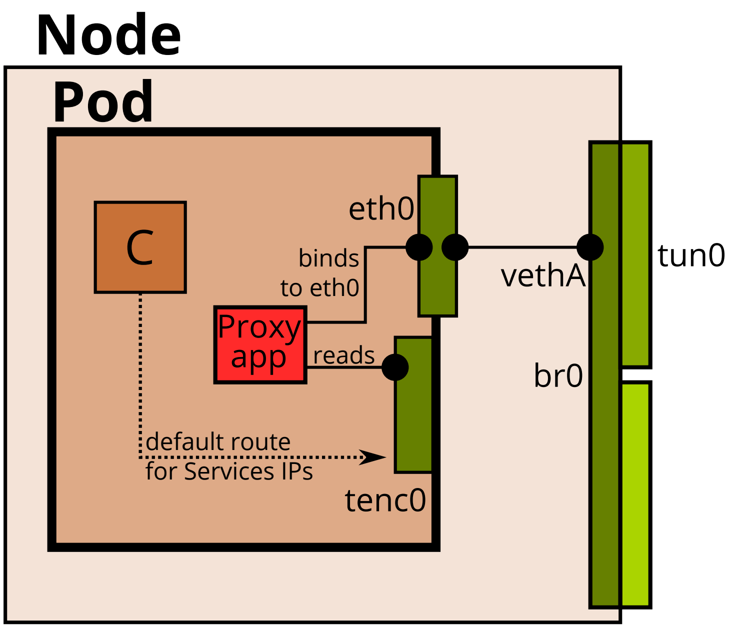

Figure 3: Overview of the proxy application implementation schema

```
FROM openshift/origin-pod:v3.11-original

USER root

RUN INST="yum install --disableplugin=fastestmirror -y" && \
    PKGS="iproute tcpdump htop iperf" && \
    $INST epel-release && \
    $INST $PKGS && \
    rpm -V epel-release $PKGS && \
    yum clean all

COPY tencrypt-proxy /opt/tencrypt/tencrypt-proxy
COPY tencrypt-setup.sh /opt/tencrypt/tencrypt-setup.sh

USER 1001
```

Listing 1: Example Dockerfile for a patched Pod, created after retagging the original image in the Minishift Docker image registry

```
with open("/etc/resolv.conf", "r") as fh:
    for line in fh.readlines():
        if line.startswith("search"):
            project = line.split(" ")[1]

while True:
    data = os.read(fd, MTU + 18)
    pkt = parse_packet(data)
    dns_info = socket.gethostbyaddr(pkt.dst)
    if dns_info[0].endswith(project):
        print("Host {} belongs to project!".format(pkt.dst))
```

Listing 2: Resolving and differentiating remote IP addresses with a Python script (using Scapy)

code would be more compatible with the upstream project and used in a dedicated OpenShift instance instead of a Docker-simulated testing environment. Since the last approach revealed some limitations of Boot2Docker and TinyCoreLinux, it was decided to reduce possible limitations by switching the Minishift VM operating system to CentOS.

Running Minishift with `--profile centos --iso-url file:///tmp/minishift-centos7.iso` creates a new profile and new VM. CentOS not only provides a full operating system with more libraries, it also offers more possibilities by installing packages with `yum`, e.g. `tcpdump`. TinyCoreLinux offers this as well with `tce-load`, but has limitations as mentioned above. `yum` could also be used inside the containers, since they are based on the CentOS base image.

Surprisingly, after setting up test-deployments and their services via the WebConsole of OpenShift, it is directly possible to use `nsenter` to enter a network namespace of a running Pod. Finding the `pid` is possible by identifying the Pod container and using `docker inspect`, extracting the `State.Pid` value. Finally, it was possible to manually edit the network adapter inside the container, because `nsenter` is executed as root from the host machine and not from within the restricted container environment.

Coming back to the original aim: transparently proxying traffic to other Services. Again, multiple approaches for traffic routing and transparent proxying were tested, starting with the configuration of IPtables rules and continuing by using virtual interfaces. In the end, a virtual TUN interface in combination with default routes and policy based routing did the trick. The solution for creating the interface and setting up default routes to a transparent proxy is implemented as shown in listing 4.

The verification of the functionality was done with a Python script which opened the TUN interface, read the TCP payload from each packet and sent the payload via a second connection towards an external host. The received answer was then passed back to the original local application which was waiting for the reply. The parsing was done with the Scapy library.

The following list summarises the approaches of this implementation step beginning with the first attempts:

1. At first, a local application should listen for connections on a local port and act as a proxy for TCP flows. For this, IPtables should do destination Network Address Translation (DNAT). This does not work as expected, because NAT rewrites the packets before passing them on according to the netfilter packet flow. The information where the packet should originally be sent to is lost and can therefore not be relayed. In this experimental stage, the IPtables `mangle` table and the `TPROXY` target were evaluated. `TPROXY` is only compatible with the `PREROUTING` chain, but packets from local applications only pass the `OUTPUT` and `POSTROUTING` chains, so `TPROXY` is not usable in this case.

2. Using a TAP interface for low-level control on link layer 2. The default route to Services must then be defined as this interface. An application opens the file descriptor to this interface via `ioctl` and reads incoming Ethernet frames from it. This worked partly, but ultimately failed because TAP interfaces need working Address Resolution Protocol (ARP) routing, since it works on layer 2. Any traffic sent towards the gateway `10.0.0.1` which was configured on the `tenc0` TAP interface resulted in ARP requests for `10.0.0.1`. Using the interface itself as a route, without any interface IP, resulted in ARP requests for the remote host address - which obviously cannot be resolved on the local interface. TAP was therefore dismissed to not introduce more complexity factors.

3. Creating the interface as a TUN interface working on the IP layer removes the need to deploy working ARP on this interface. The application reading packets from the interface receives IP packets only. Configuring the TUN interface as the default gateway for traffic towards the Services IP range works transparently for applications running inside the network namespace. A problem occurs as soon as the application reading the packets wants to relay/proxy the received packets. The default route will be applied to this application as well, resulting in a traffic loop.

4. To solve this problem, two interface IP addresses needed to be added to the TUN interface, `10.0.0.2/24` and `10.0.0.3/24`. The default gateway for new traffic being the `10.0.0.1/24`, routing via `tenc0`. At this point the IPtables rules and the usage of multiple routing tables were re-introduced. An application which uses this route uses the local address `10.0.0.2` as the packet sender. The proxy application should then receive the packets on the virtual interface, rewrite the source IP address and re-send the packet. The `mangle` table should `MARK` packets which were sent from `10.0.0.3`, and the `fwmark 2 table 3` rule should route those packets via another interface. The whole combination seemed plausible, but failed, because packets were sent via the `tenc0` interface again. Binding to a specific interface did not work in the Python script.
5. The solution for this setup was a variation of the previous step. The `10.0.0.3/24` interface address and all IPtables rules were removed, only routing policies were still applied. The proxy application reads from the TUN interface and binds not as `10.0.0.3`, but as `172.17.0.X` on the `eth0` interface. This combination lets all default traffic originated in local applications proceed according to the default routing table, but marks packets `from 172.17.0.X/16` to use another routing table which uses `172.17.0.1` as the default gateway. See listing 4 for the commands needed.

4.4.2. Part 2: Differentiation of Project-internal and -external traffic flows

OpenShift makes heavy use of the internal DNS hostnames and up-to-date resolvers in combination with the usage of the Virtual IP (VIP) address space. This feature was focus to testing the possibility of differentiating traffic flows, looking for a way to utilise the DNS to identify remote IP addresses within the same Project namespace and the rest.

Looking at the DNS config inside a container, we see that OpenShift uses a hierarchical DNS structure which starts its search for hostnames in the Project DNS zone.

```
nameserver 172.30.0.2
search myproject.svc.cluster.local svc.cluster.local cluster.local
```

Listing 3: The contents of a Pods `resolv.conf`

This means resolving requested service addresses could identify connections which should be encrypted. What is missing is a possibility for a Pod to look up its own Project namespace name. The container environment variables of a running application offers the variable `OPENSHIFT_BUILD_NAMESPACE` with the name, but this variable is not available in the Pod container, which is the main target.

Given the information contained in the `resolv.conf`, the `search` domain can be utilised to lookup and differentiate host names of remote IP addresses. The host names are queried by reverse IP lookups. To test this setup, the environment contained two Projects, with two Services in the first and one in the second. One important thing to know: the DNS resolver is also in the `172.30.0.0/16` address range and must be excluded from the default route which is routed over the TUN interface.

Utilising a Python script again, this step can be done by reading the `resolv.conf` and then issuing reverse lookups for remote IP addresses. Listing 2 shows extracted code from the working implementation. Of course, this tool should later be implemented in Go and use a cache to limit the reverse lookup query count.

4.4.3. Part 3: Encryption of traffic

At this point we have a virtual interface as default gateway on which packets can be intercepted and modified. The proxy application has the possibility to differentiate between remote hosts which belong to the same Project and Project-external hosts. The application listening on the virtual network adapter therefore should know which IP addresses are remote Project-internal services and encrypt accordingly. It would now be possible to encrypt payloads of TCP streams by reading the original contents, buffering them and then forward them encrypted. The remote Pod would have the same setup, receiving encrypted packets and decrypting them to pass the original payload to the local application.

But there is one element missing: how does a Pod know it received encrypted traffic and how can encrypted traffic be routed through a decrypting proxy application? Some ideas that came to mind:

- The policies applied for outgoing traffic should be applicable for incoming traffic as well. This way, different routes can be set up.
- As a fallback, IPtables `MANGLE` and `TPROXY` (including the `IP_TRANSPARENT` socket option) might be applicable. In this case, the need for transparently proxying incoming traffic, is indeed solvable by routing traffic through a listening application. The application should work the same as the TUN reading application, only in reverse.
- In case of tunneling techniques, SOCKS5 could also take over establishment of transparent sockets.
- As soon as the problem of routing external traffic into a proxy application is solved, incoming traffic can be differentiated the same way as explained above. An application which receives traffic flows from a Project-internal IP should expect this traffic to be encrypted.
- The previous point could even be extended to make the receiver *require* encrypted traffic. Given that most internal traffic is based on TCP/HTTP, security could be enhanced by blocking traffic from unpatched Pods.

The results of the proof of concept implementation are given in the next section.

4.5. Proof of concept implementation

At this point we make a cut, ending the thought process of possible approaches and jump to the implementation results. The proof of concept (PoC) implementation was written in the Go language with the following components:

- A DNS upstream proxy which reads requests and parses them to extract information.
- A TUN interface handler, reading packets from applications and writing received replies.
- UDP encapsulation of payloads with encryption and decryption functionality, using a listener on a specified port for Tencrypt UDP packets.
- Raw sockets to route traffic towards targeted Services listening on local interfaces.

For an overview over the traffic flow, see the flow graph in figure 5. There were multiple obstacles to be taken, which are described in the following paragraphs.

The first approach in finding out if a host is internal or external – as mentioned in section 4.4.2 – lead to one failed request at the beginning of a connection establishment. An application inside a Pod requests the IP address of a remote Service from the DNS server, receives the IP and sends the TCP SYN packet towards the host. The packet is routed over the `tenc0` interface, but can not directly be handled, because the proxy would have to examine the state (internal/external) of the requested host via reverse lookup. It was necessary to change this approach and go one step further.

For optimisation, external hosts should be white-listed so traffic targeting these hosts would not flow through the proxy. This and the caveat mentioned previously lead to the implementation of a DNS proxy inside the Pod. This DNS proxy opens a DNS socket on the loopback interface and the Pod gets this „new" DNS server assigned as the default nameserver in `/etc/resolv.conf`. All requests to this socket are routed to the upstream DNS server on a single, static UDP connection (this might even lead to an improved component on its own, reducing the DNS connections inside the network). Answers from the upstream DNS are inspected and parsed, replies are matched to clients by the DNS ID field. In case the host is a Pod outside the namespace of the requesting Pod, the IP is added as a static route over `eth0`. This needs further tweaking, as IP addresses might change and routes must be deleted in case a host receives a new VIP.

Even though in the fundamentals (section 4.1), which were defined before the practical investigation, stated that only application data would be encrypted, the PoC does indeed encapsulate the whole packet received from client and service applications. This step was done due to failing handshakes in TCP sessions in the case of payload encryption and reassembly of the TCP packet. Further development of the proxy application might be able to pick up this approach again.

As mentioned, the `tenc0` TUN interface is one central point of exchange for proxied applications. But during the implementation phase it came to light that the TUN interface is not enough to interact with applications listening on the public interface of a Pod. Example: a Pod offers a web service on port 8080, the web server listening for connections is bound to the public interface. Packets sent to the `tenc0` interface however are not read by this web server application, because there is no listening socket waiting for packets. A second channel is needed to push incoming packets onto the kernel network stack like a „normal" client would do when connecting. This is solved by using a raw socket inside the target Pod as a sending endpoint. The raw socket uses `10.0.0.2` to send the received payload towards the listening service.

The whole procedure of reading, encapsulating and unpacking has one other problem to be solved: a Service is addressed by its Virtual IP (VIP), not by the IP the Pod uses on its public interface. Meaning, if a client application in Pod A resolves `myservice.myp.svc` to `172.30.123.123`, the packets would be tunneled over the Tencrypt proxy. But as soon as the packet is reassembled on the remote Pod B, B does not know about the virtual IP address, because the virtual IP addresses are translated by NAT rules on the `docker0` bridge. Sending the packets to the Tencrypt UDP listener reachable over the virtual IP is no problem, these packets are rewritten as expected. Packets which are received as payload by the Tencrypt UDP endpoint have to be translated as well. This feature is implemented in Tencrypt, too, including the necessary re-calculations for IP and TCP checksums.

4.6. Throughput measurements

Once the PoC could be verified to successfully proxy TCP connections, e.g. a request made by `curl` inside the client Pod to a web service offered as a Service on a remote Pod, throughput measurements should show how much performance the proxy connections would lose. The measurements were taken with `iperf` inside the Minishift VM (Virtualbox, 2 CPUs, 2GB RAM). To run `iperf` between the Pods the IPtables rules had to be extended to allow TCP traffic as well. A server was then started in the network namespace of the first Pod, the client in the second. The collected measurements were exported as CSV and parsed with a Python script. The results are displayed in figure 4.

The first three runs were done without applying the patches to the Pods. Different amounts of clients invoked by iperf did not show any big differences in performance. Looking at the test runs with applied patches and running proxy however, one can see that the amount of clients make a difference. It is observable that each client has a limited bandwidth it can use

Test description	Transmitted	Throughput server	Throughput client
no patch, 1 client	22640.38 MB	2262.3 MB/s	2263.83 MB/s
no patch, 5 clients	27350.0 MB	2724.02 MB/s	2727.22 MB/s
no patch, 10 clients	25817.38 MB	2410.15 MB/s	2538.02 MB/s
patched, 1 client	2.46 MB	0.01 MB/s	0.16 MB/s
patched, 5 clients	12.29 MB	0.06 MB/s	0.82 MB/s
patched, 10 clients	24.58 MB	0.13 MB/s	1.64 MB/s
patched, 20 clients	24.86 MB	0.12 MB/s	3.27 MB/s

Figure 4: Results of iperf measurements with different amounts of clients

for encrypted traffic, so using multiple clients in parallel result in the addition of these limits.

Another factor which was important to be evaluated was the payload encryption. Watching the traffic on the `docker0` bridge on the host showed that `iperf` uses random numbers as payloads in packets. These were encrypted in the fourth to seventh runs as expected. Further evaluation of the results can be found in the conclusion in the next section 5.

5. Conclusion

Tencrypt showed that transparent encryption of Pod-to-Pod traffic with regard to the differentiation of Project-internal and -external Pods in an OpenShift environment is possible.

In section 3 the security requirements and the threat model were gathered. We have seen that an implementation which would run in production infrastructure would need to fulfil the needs for authentication, integrity, confidentiality and possibly availability and authorisation. Even though the malicious access to the Node host and its networking cannot be prevented by Tencrypt, the concept of Tencrypt offers a practical component in Node security regarding future developments in container hardening techniques such as hardware-based isolation of memory.

Sections 4.1 and 4.2 gave an overview of the basic conceptual assumptions of Tencrypt and explored multiple alternatives of implementation. Additionally, possible future problems of each alternative were taken into consideration.

The Minishift development environment was examined in detail in section 4.3. We looked at the network configuration, the modes of Secret management, how the Docker daemon could be interacted with and how the internally deployed Pod image could be patched to include changes needed to execute Tencrypt in namespaces.

Going further, section 4.4 covered three parts of practical implementation details. Part one highlighted the diverse possibilities in network configuration, showing which parts could be covered with IPtables and policy based routing and the usage of TUN and TAP interfaces. The differentiation of Project-internal and -external traffic was demonstrated in part two. The section was concluded with part three which provides ideas regarding the encryption and decryption of packets.

Section 4.5 was a wrap-up of the actual proof of concept implementation written in Go. It includes a DNS proxy, the TUN interface reader-writer, the UDP encapsulation service, encryption and decryption of encapsulated packets and the local use of raw sockets.

As a final step, the implementation was tested regarding the throughput performance. Examining the measurement results in section 4.6, we can see that the throughput is very low compared to unpatched connections. It should also be noted that the PoC fails the requirements listed in section 3, because the tunnel only applies AES encryption of the payload without any promises regarding authentication or integrity. These are topics for further development iterations, because the code is intended to be an unoptimised proof of concept. Nevertheless, Tencrypt shows that the patches work as intended and that the proposed concept proved to be a possible candidate to be integrated into the OpenShift software stack.

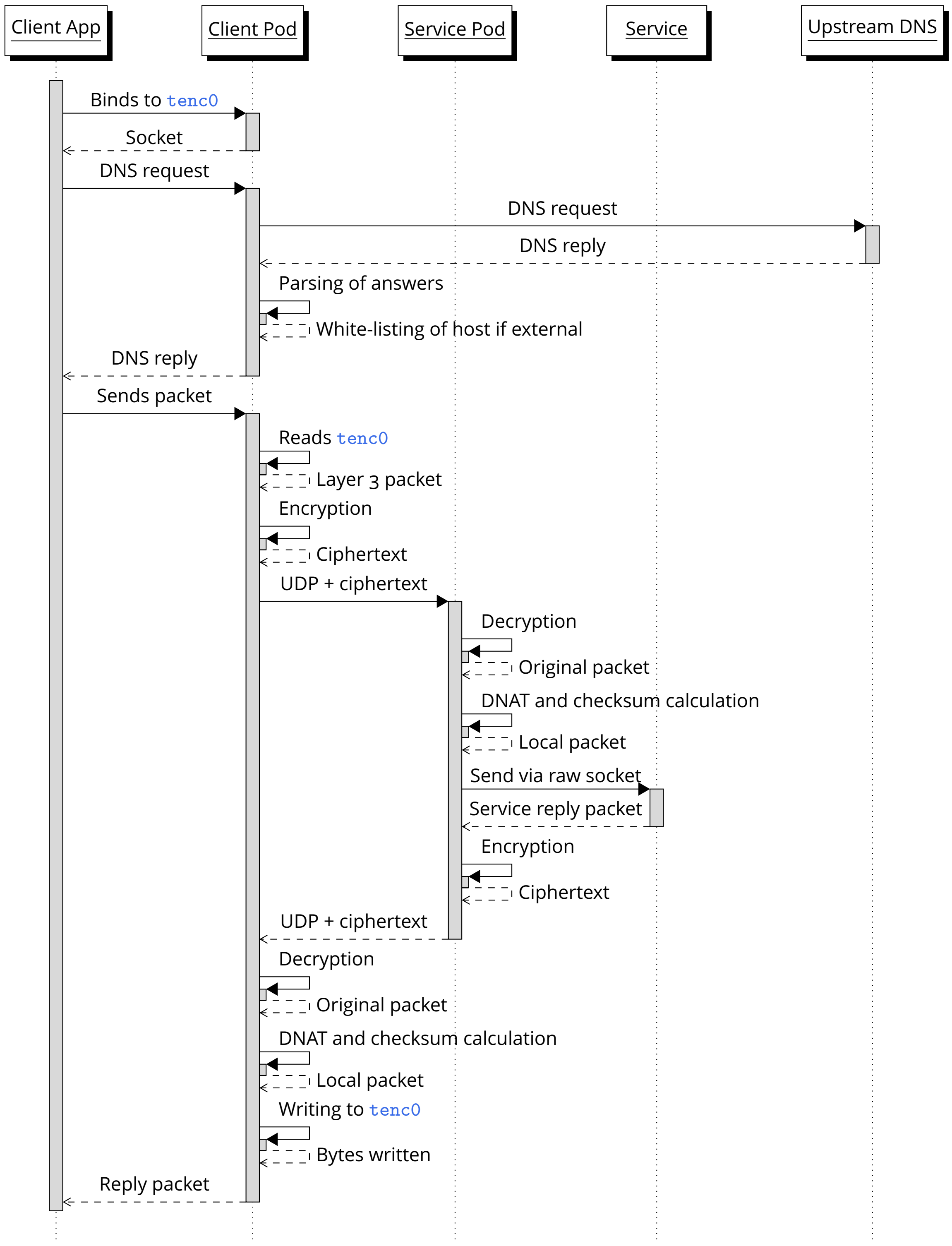

Figure 5: Flow graph displaying sending and receiving proxied packets

```bash
#!/bin/bash

# Create TUN
mkdir /dev/net && mknod /dev/net/tun c 10 200 && \
ip tuntap add mode tun tenc0 && \
ip link set tenc0 up
ip link set mtu 1440 dev tenc0
ip addr add 10.0.0.2/24 dev tenc0

# Create routing policy
OWNIP=$(ip a show dev eth0 | grep "inet " | awk '{ print $2 }')
ip rule add from $OWNIP lookup 2

# And the routes
ip route add 172.30.0.0/16 via 10.0.0.1 dev tenc0
ip route add 172.30.0.2/32 dev eth0
ip route add default via 172.17.0.1 dev eth0 table 2

# Prevent TCP RESETs on raw sockets issued by kernel
iptables -I OUTPUT -s 10.0.0.2 -p tcp --tcp-flags RST RST -j DROP

# Change DNS server to local DNS proxy
echo -e "nameserver 127.0.0.1\nsearch myproject.svc.cluster.local svc.cluster.
    local cluster.local\noptions ndots:5" > /etc/resolv.conf
```

Listing 4: Setup script to be run before proxy execution as root with nsenter

```
# docker image ls
REPOSITORY                              TAG                 IMAGE ID
172.30.1.1:5000/project2/cakephp-ex     latest              6da36eca23b1
172.30.1.1:5000/myproject/django-ex     latest              cbae17d2ceb4
172.30.1.1:5000/myproject/nginx-ex      latest              1e79d27e9dda
openshift/origin-control-plane          v3.10               57543ab622d1
openshift/origin-hypershift             v3.10               4d95bae07cf6
openshift/origin-pod                    v3.10               1611d858742d

# docker image tag 1611d858742d openshift/origin-pod:v3.10-original
# docker build -t openshift/origin-pod:v3.10 ../implementations/origin-pod/
# docker image ls
REPOSITORY                              TAG                 IMAGE ID
openshift/origin-pod                    v3.10               fb9137adab6e
172.30.1.1:5000/project2/cakephp-ex     latest              6da36eca23b1
172.30.1.1:5000/myproject/django-ex     latest              cbae17d2ceb4
172.30.1.1:5000/myproject/nginx-ex      latest              1e79d27e9dda
openshift/origin-control-plane          v3.10               57543ab622d1
openshift/origin-hypershift             v3.10               4d95bae07cf6
openshift/origin-pod                    v3.10-original      1611d858742d
```

Listing 5: Terminal window for renaming the Docker origin-pod image.

```bash
#!/bin/bash

OC_EXE=/var/lib/minishift/bin/oc

$OC_EXE login localhost:8443 --insecure-skip-tls-verify=true --username=
    developer --password=123

# Flush the TENCRYPT chain
iptables -t nat -F TENCRYPT

# If the flush reported exit code 1, the chain probably does not exist yet
if [ $? -eq 1 ]; then
    # Create the chain
    iptables -t nat -N TENCRYPT
    echo "Created TENCRYPT chain"

    # Insert a rule at the beginning of the PREROUTING NAT chain to direct
    # UDP traffic for port 1337 into the TENCRYPT chain
    iptables -t nat -I PREROUTING 1 -p udp -m udp --dport 1337 -j TENCRYPT

    # Same for TCP (for e.g. iperf)
    iptables -t nat -I PREROUTING 1 -p tcp -m tcp --dport 1337 -j TENCRYPT
    echo "Inserted PREROUTING rule for chain TENCRYPT"
else
    echo "Flushed existing TENCRYPT chain"
fi

# Iterate over all affected Services by name
for SVC in "nginx1" "nginx2"; do
    # Get the currently allocated real internal IP address
    IP=$($OC_EXE describe svc/$SVC  | grep 'Endpoints:' | awk '{ print $2}' |
    cut -f1 -d ":")

    # Get the virtual IP address for this Service
    VIP=$($OC_EXE describe svc/$SVC  | grep 'IP:' | awk '{ print $2}')

    # Add the NAT rules to allow Tencrypt traffic for each Service
    iptables -t nat -I TENCRYPT -d $VIP/32 -p udp -m udp --dport 1337 -j DNAT --
    to-destination $IP:1337
    iptables -t nat -I TENCRYPT -d $VIP/32 -p tcp -m tcp --dport 1337 -j DNAT --
    to-destination $IP:1337

    if [ $? -eq 0 ]; then
        echo "Routed TCP/UDP traffic on port 1337 towards $VIP to $IP"
    fi
done
```

Listing 6: IPtables script used to allow traffic on port 1337 between specified Pods

Appendices

A. Components of OpenShift

Kubernetes Originally developed by Google, Kubernetes is a platform for management of containers in Cloud architectures. Infrastructure components (hardware servers, virtual machines) participating in a Kubernetes cluster are called **Nodes**. Nodes are controlled by a **Master** and host multiple **Pods** and each Pod consists of multiple **Containers**.

Container runtimes

Docker Most popular container toolchain. Started as a collection of tools to better utilise Linux cgroups and namespaces, based on LXC. Later grew out to use its own API (`libcontainer`). Uses „images" and file systems like OverlayFS to easily share and stack application container configurations. The Docker Hub serves as a public registry for pre-configured images.

CRI-O Container Runtime Interface for OCI (CRI-O), optimized for Kubernetes.

rkt Container runtime and image standard developed by CoreOS for their Container Linux distribution.

Networking

Software Defined Networking (SDN) OpenShift supports the integration of different SDN provider plugins. The default stack uses Open vSwitch for SDN configuration.

Open vSwitch (OVS) An open source distributed virtual multi-layer switch with an extensive feature set (STP, VLANs, bonding, LLDP, Policies, IPv6, IPsec, VXLAN and more).

Container Networking Interface (CNI) The plug-in interface for networking in Containers (OpenShift Plugin, Flannel, Nuage, Calico, Contrail, Cisco Contiv, Big Switch, VMware NSX-T).

Logging „EFK", standing for:

ElasticSearch Full-text search engine based on Lucene, written in Java.

Fluentd Data collection and event log analyser written in Ruby.

Kibana Visualisation UI plugin for ElasticSearch.

Monitoring

Prometheus Monitoring and alerting toolkit originally developed by SoundCloud, now a community project of the CNCF.

Kubernetes Metrics Server metrics collector, formerly called Kubernetes Heapster.

Hawkular Metrics Storage engine for metric data using Cassandra as storage back end.

Apache Cassandra Decentralised scalable and fault tolerant database.

etcd Distributed fault-tolerant key-value store.

SkyDNS Distributed service announcement and discovery service built with etcd.

Istio Service Mesh to connect, manage and secure microservices.

Open Service Broker API Specification developed by multiple cloud native platform providers to define API endpoints for cloud software an SaaS.

B. Linux components

B.1. Namespaces

Namespaces wrap existing global system resources into isolated abstractions which are only visible to processes which are members of this same namespace. Uses the `/proc/[pid]/ns/` hierarchy for API files. Uses the `clone`, `setns` and `unshare` system calls. System calls might need the `CAP_SYS_ADMIN` capability, except for user namespaces. Entering a namespace can be done with `nsenter` and flags. See the `man 7 namespaces` man page.

cgroup cgroups controlling processes in this namespace.

ipc System V IPC, POSIX message queues.

net Network stack (devices, interfaces, routing tables, connections, ports).

mount Contains file system mount points.

pid Contains an isolated list of process identifiers of processes running in this namespace.

user User and group identifiers. Means, e.g. a process can have an unpriviledged user ID on the system, but run forks as root inside the namespace.

uts Contains the hostname namespace.

B.2. Linux Control Groups

Provided through the `cgroupfs` pseudo-file system, control groups offer an API to group processes in hierarchical groups. Resource usage of each group is then monitored and limited. Groups in lower hierarchy levels cannot exceed their resource usage above the parent limitations. Each resource type is implemented as a `subsystem`, also known as `controller`. Since Linux kernel version 4.5, cgroups v2 is available in the mainline. See the `man 7 cgroups` man page. Below is a list of available controllers in cgroups v2:

io Controls and limits access to specified block devices. IO control can be enforced by throttling and upper limits. `blkio` in v1.

memory Reports and limits process and kernel memory, as well as swap, used by the cgroup.

pids Limits the number of processes which can be created in the cgroup.

perf_event Allows the usage of `perf` on certain events (e.g. system calls).

rdma Limits the use of Remote Direct Memory Access (RDMA)-specific resources per cgroup.

cpu Provides limits for CPU scheduling and accounting of CPU usage, as well as an API to guarantee a specified minimum CPU share for this cgroup in case the system is busy.

B.3. SELinux

A Linux kernel module providing improved security mechanisms through Mandatory Access Control (MAC) and policies. These policies include process management, access to the file system and network sockets and the usage of capabilities and are appliable to users and processes. SELinux improves container security by offering more security mechanisms. Used by default on RHEL, CentOS and Android.

B.4. Secure Computing Mode (seccomp)

Processes can call the `seccomp` system call with either the `SECCOMP_SET_MODE_STRICT`, the `SECCOMP_SET_MODE_FILTER` or the `SECCOMP_GET_ACTION_AVAIL` arguments. The first two restrict the set of allowed system calls for the calling process.

C. Sources, References, List of Figures, Glossary

References

[Sho14] Adam Shostack. *Threat modeling designing for security*. J. Wiley & Sons, 2014. ISBN: 9781118809990.

[Mou16] Adrian Mouat. *Docker Security - Using Containers Safely in Production*. O'Reilly Media, 2016. ISBN: 9781492042297.

[Ric16] Wolfram Richter. *OpenShift Container Netzwerk aus Sicht der Workload*. See https://people.redhat.com/~llange/OCP_Netzwerk.pdf. 2016.

[Ros16] Rami Rosen. *Namespaces and cgroups, the basis of Linux containers*. See https://www.netdevconf.org/1.1/proceedings/slides/rosen-namespaces-cgroups-lxc.pdf. Feb. 2016.

[PHO17] Stefano Picozzi, Mike Hepburn, and Noel O'Connor. *DevOps with OpenShift*. See https://www.openshift.com/devops-with-openshift/. O'Reilly Media, 2017.

[Hat18] Red Hat. *Openshift Container Platform Technical Overview*. See https://www.openshift-anwender.de/wp-content/uploads/2018/04/OpenShift-Technical-Overview.pdf. Apr. 2018.

[Hau18] Michael Hausenblas. *Container Networking: From Docker to Kubernetes*. O'Reilly Media, Apr. 2018.

[MC18] Veer Muchandi and Shanna Chan. *Network Security for Apps on OpenShift*. See https://www.redhat.com/files/summit/session-assets/2018/Network-security-for-apps-on-OpenShift.pdf. 2018.

Online resources

[Low09] Scott Lowe. *What is SR-IOV?* Dec. 2, 2009. URL: https://blog.scottlowe.org/2009/12/02/what-is-sr-iov/.

[Huq18] Ariful Huq. *OpenShift Commons Briefing #131 End-to-End Security for Microservices*. June 14, 2018. URL: https://www.youtube.com/watch?v=yzVc06AJ89s.

[Blo] Open Cloud Blog. *Linux Switching – Interconnecting Namespaces*. URL: http://www.opencloudblog.com/?p=66.

[Bri] Davide Brini. *Tun/Tap interface tutorial*. URL: https://backreference.org/2010/03/26/tuntap-interface-tutorial/.

[OKDa] OKD. *Container Security Guide*. URL: https://docs.okd.io/latest/security/index.html.

[OKDb] OKD. *OpenShift SDN*. URL: https://docs.okd.io/latest/architecture/networking/sdn.html.

[OKDc] OKD. *Security Context Constraints*. URL: https://docs.okd.io/latest/admin_guide/manage_scc.html.

[sec-road] *OpenShift Roadmap Team Security*. URL: https://ci.openshift.redhat.com/teams_overview_security.html.

List of Figures

List of Listings

Glossary

Control Plane Protocols and data tables used to collect and calculate the topology of a network. 25

Data Plane Routing of packets based on routing information collected via the Control Plane. 25

Origin Kubernetes Distribution (OKD) Upstream Kubernetes distribution embedded in Red Hat OpenShift, also known as OpenShift Origin. 4, 25

Software Defined Networking (SDN) Network architecture with separated Data and Control Planes (using centralized controllers). 4, 22, 25

Acronyms

ACI Application Centric Infrastructure. 7, 25
ARP Address Resolution Protocol. 14, 25

CD Continuous Deployment. 4, 25
CI Continuous Integration. 4, 25
CNCF Cloud Native Computing Foundation. 25
CNI Container Networking Interface. 4, 22, 25
CRI-O Container Runtime Interface for OCI. 22, 25

ECMP Equal-cost Multipath. 7, 25
ETSI European Telecommunications Standards Institute. 25

HA High Availability. 4, 25

IETF Internet Engineering Task Force. 25

MAC Mandatory Access Control. 23, 25

NAT Network Address Translation. 14, 25

OCI Open Container Initiative. 25
ONF Open Networking Foundation. 25
OVS Open vSwitch. 5, 22, 25

PV Persistent Storage. 5, 25

RDMA Remote Direct Memory Access. 23, 25
RHEL Red Hat Enterprise Linux. 4, 25

S2I Source-To-Image. 25
SCC Security Context Constraint. 12, 25
SPIFFE Secure Production Identity Framework for Everyone. 6, 25
SR-IOV Single Root I/O Virtualization. 7, 25

TCO Total Cost of Ownership. 25

UAT User Acceptance Testing. 25

VIP Virtual IP. 10, 15, 17, 25
VM virtual machine. 4, 7, 10, 22, 25
VNID Virtual Network ID. 5, 25