ISOVALENT
now part of cisco

Cilium Network Policy Deep Dive

Authors

Jonathan Stringer is a Principal Software Engineer at Isovalent with over ten years of experience building efficient software networks. As a contributor to open source projects ranging from Open vSwitch to Cilium and the Linux kernel, Joe has built critical components to enforce stateful firewalling and security policies across both cloud and traditional environments. Joe is a frequent public speaker with multiple patents, serves as a Co-Maintainer for the Cilium project, and he is a member of the eBPF Foundation Steering Committee. In his spare time, Joe enjoys travel, cycling, music and gaming.

Nicholas Lane is a Principal Solutions Architect at Isovalent focusing on customer success. Nicholas has over a decade of experience as a customer success engineer helping users adopt cloud native technologies. Throughout his career Nicholas has focused on enabling users to start using new technologies with as little friction as possible. As a former member of the Kubernetes release team, and an active member of the Kubernetes community, Nicholas has experience working with complicated open source projects and is an avid supporter of open source technologies at large. Outside of customer success Nicholas enjoys watching speed-runs of video games with his wife, cycling, woodworking, and gaming whenever there is time.

With special thanks to the reviewers of this eBook:

Bill Mulligan
Casey Callendrello
Chris Tarazi
Dean Lewis
Jarno Rajahalme
Julian Wiedmann
Marcel Zięba
Piotr Jabłoński

Index

Introduction

The security of networks is critical for the protection of data distributed and consumed by modern applications. While there are many varied aspects to implementing a secure environment, a key aspect is in observing and securing the communications that occur between applications in an environment and with remote peers: network policy enforcement.

This paper intends to provide a picture from the ground up about how Cilium approaches network policy enforcement to align the *intent* of platform operators with the *realized state* in the environment that implements that policy. This information is provided in a layered approach: beginning from the high level abstractions and considerations for successful association of *intent* to *location* of endpoints who communicate; then by exploring the policy model that Cilium implements from the cluster-wide level through to per-node, per-policy, per-workload and per-packet granularity to apply the policies to a set of *subjects* and *peers* that communicate; and finally by exploring the policy language itself and the deployment considerations for scaling and monitoring Cilium with network policies.

The goal of this document is to educate platform operators, security experts, and SREs about how to operate Cilium network policies in production. Many of the insights described in this document are generally applicable to all versions of Cilium, however some functionality described herein is currently only available in Isovalent Enterprise for Cilium. Where relevant, footnotes describe the distribution of Cilium, product version numbers and timelines with links to either the Cilium or Isovalent Enterprise for Cilium documentation where you can learn more about the features.

You should come away from this with a solid grasp of Cilium's modeling of *Security Identity*, how those Identities are associated with workloads and expressed in network policy artifacts, and how to monitor the operation of Cilium's network policy enforcement in production environments.

Background & Concepts

The primary intent behind writing an effective network policy is to establish sets of applications which are allowed to talk to one another, or in other words to define *who* can talk to *whom*. In a world of increasingly ephemeral cloud-native applications, the ability to tightly bind important metadata to applications and enforce policies using that metadata is crucial to building a secure and performant environment. Cilium achieves this through a notion of a *Security Identity*, the context that describes the boundary between one application and another. All other parts of the policy engine from policy language semantics to the deep details of the Cilium eBPF dataplane follow from this core concept. The Security Identity is used to select *subjects* for a network policy and the remote peers that the subject can communicate with. Security Identities also enable the classification of communications with *peers* outside of the cluster into groups that are meaningful for restricting network access.

Policy intent may be defined by multiple resource types: Kubernetes Network Policies (KNP), Cilium Network Policies (CNP) and Cilium Clusterwide Network Policies (CCNP). Regardless of which type is used, all network policies specify the subject, a set of peers, optional higher-level predicates, and a specific policy action which informs Cilium whether to allow or deny traffic.

Label-based Security Identity

Modern application deployment mechanisms support enriching workloads with metadata to define that workload's attributes - such as the workload's role, environment, or software release. Cilium generalizes this notion as a set of *labels* made up of a source, key, and value.

For example, if a Kubernetes Pod has the label `app.kubernetes.io/name: mysql`, Cilium combines the Kubernetes source with the key and value of this label, and expresses the label in the form `k8s:app.kubernetes.io/name=mysql`. The source identifies where the label comes from, which can be either internal to Cilium or from an external system such as Kubernetes, where each system specifies a set of key-value pairs. A workload may have many keys, but each unique key from a source has just one specific value. Collectively, the set of labels represents *who* the workload is from a policy perspective. Cilium calls this the Security Identity.

Sources of Label Information

Cilium supports a variety of sources for label information, the most common of which is the Kubernetes control plane. Cilium may also learn labels through node-local sources such as a local DNS proxy or the networking stack. Individual hosts running Cilium can inform one another about the addresses used by the nodes in the cluster by publishing this information into a central data store[1]. The Cilium Agent on each node is responsible for gathering the labels for an endpoint and assembling them into a canonical form for use in network policy enforcement on the node. This process involves fetching the labels, filtering them to labels that are considered *Identity-Relevant* (that is, relevant to enforcing network policy), and allocating a Security Identity for the label set.

Identity-relevant Labels

Depending on application usage of different label keys and values, the unique set of labels used within a cluster can be very large. However, not all such labels are important to the security posture of those applications. For instance, you may find it meaningful to express that workloads with the label `app.kubernetes.io/name=client` may communicate with workloads with the label `app.kubernetes.io/name=server`. On the other hand, it may be less useful to express who may communicate with workloads with the label `kubernetes.io/arch=amd64`. In large environments, filtering the set of security-relevant labels is crucial to ensuring that network policies are effective at targeting workloads, as well as scaling the distributed network policy enforcement efficiently. The later section **Limit Label Cardinality** explores how to configure this option in your environment.

Reserved Labels

Cilium assigns a set of "reserved" labels[2] to various in-cluster components such as the local node (`host`), remote nodes (`remote-node`), and the Kubernetes control plane (`kube-apiserver`). There are other optional features with their own reserved labels, such as Cilium's implementation of Kubernetes Ingress (`ingress`) and the cluster connectivity health checker (`health`). The reserved labels `init` and `unmanaged` should typically not be seen in a cluster with a fully Cilium-managed network, as they are assigned to workloads when Cilium has incomplete information and is otherwise unable to associate a more specific set of labels with the workload.

Finally, network locations outside of the Cilium managed clusters are assigned the world label. Any particular layer 3 network location may be associated with multiple labels. For instance, if the Kubernetes Control Plane resides outside the current cluster, it may have both the `kube-apiserver` and `world` labels.

[1] As of 2024, Cilium supports distributing information between nodes via the Kubernetes control plane or via a dedicated Etcd cluster.
[2] Each reserved label has a corresponding **special identity**[3].

Security Identity

The Security Identity of a network endpoint is a set of labels with a corresponding number. Cilium assigns a Security Identity to each network endpoint and uses the Security Identities to enforce network security postures between network endpoints, such as to allow or deny traffic between the Security Identities. When you create a workload in a Cilium-managed environment, Cilium provisions the network dataplane before the application runs. During this network initialization phase, Cilium assigns a number for the Security Identity and associates this number with the Identity-Relevant labels of the workload. On each node in the cluster, Cilium reuses the same Security Identity when additional instances of the same workload are created in the cluster. The motivation for this definition of a Security Identity is two-fold: Firstly, for a Security Identity to have meaning to infrastructure operators, it is useful to describe it in terms of the labels of the corresponding workloads. Secondly, in order to make the Security Identity efficient for machines to process and enforce network policy, it is helpful to have an integer which represents the same security boundary.

Security Identities for Local Workloads

Cilium acts as a Container Network Interface (CNI) plugin in the cloud-native environment. As part of this responsibility, Cilium creates the network interfaces for the workload and configures the network forwarding behavior and security posture. In order to tightly bind the Security Identity to the workload, Cilium embeds the Security Identity into eBPF programs, loads the programs into the operating system kernel and attaches them to the network interface responsible for network connectivity for the workload. These eBPF programs handle all network traffic flowing to or from the workload, so Cilium is perfectly positioned to ensure that the workload's security posture is locked down according to the installed network policies. Any traffic entering or leaving that workload must pass through Cilium's network enforcement layer before the traffic is routed towards its destination.

Security Identities for Remote Peers

For network endpoints that are not local to the cluster, Cilium gathers the set of labels associated with that network endpoint via a range of user configurations or service discovery mechanisms. Using this set of labels Cilium allocates a Security Identity for the external endpoint. The external endpoint's Security Identity number is allocated from a range dedicated to endpoints that are external to the cluster. In contrast with typical workload Security Identities described above which apply to all instances of the same workload in a cluster, remote Security Identities often refer to a larger group of network locations outside the cluster. In the most extreme case, this can be used to categorize all traffic outside of the cluster - known as the World Identity. If Cilium does not otherwise have more specific information about the labels for a network endpoint it will fall back to considering the endpoint as having this World Identity. This **World Identity** represents the least trusted group of peers that a workload may communicate with.

Selectors

A Selector is a query expression over sets of labels. Selectors are used to define subjects and peers of network policies by their labels. Cilium supports various selector expressions including the `matchLabels` statement which selects a particular label with a specific key and value, and `matchExpressions` that can select labels using more complex set operations. Expressions can select by the lack of a specific label, or by a label key having a value within a predetermined set. These selectors follow the standard **Kubernetes label selector semantics**[4].

```
endpointSelector:
  matchLabels:
    app.kubernetes.io/name: client
```

An example subject selector matches workloads with the label key “app.kubernetes.io/name” and value “client”.

When you specify subject selectors in a network policy, this determines which workloads will enable network policy enforcement. This may include enforcement when network traffic egresses or ingresses the endpoint’s network layer, depending on the rules specified in the policy. The rules also leverage selectors to define the peer of a policy: the destination of a request in the case of egress policy, or the source of a request in the case of ingress policy. Cilium peer selectors may select Cilium-managed Endpoints, Cilium-managed Nodes, Entities, IP Prefixes, FQDNs, Groups, or Services.

Subject Selectors

Cilium Network Policies have a **Subject Selector** which determines the workloads affected by the policy. This enforcement point is defined by the `spec.endpointSelector` or `spec.nodeSelector` fields in a Cilium Network Policy or Cilium Clusterwide Network Policy. Every supported network policy type can select Endpoints as subjects for the network policy, but only `CiliumClusterwideNetworkPolicy` supports selecting Nodes as a subject for a network policy.

When the Cilium Agent receives a network policy, it processes these fields to determine which workloads the policy applies to. The rest of the network policy object will only be processed and applied if there is a local workload selected by this subject selector. These subject selectors use `matchLabels` or `matchExpressions` statements to select endpoints or nodes by their labels. Other selector types described below are not supported for subject selectors.

Subject Nodes

When using a Node Selector for the subject selector, Cilium enforces the network policy for the host itself. This feature is called the Host Firewall, and it applies network policy enforcement for traffic that is being sent from or received towards applications running directly on the host itself. In a Kubernetes environment, any Pod that has the property `hostNetwork: true` is considered to be running as part of the underlying Node and is subject to Node Selectors rather than Endpoint Selectors. Similarly, all applications running directly on the host itself outside of a container or VM are subject to the host firewall. In a Cilium-managed environment, all of these applications share a common network policy enforcement point. Similar to the Endpoint selector, the Node selector also supports both `matchLabels` and `matchExpressions` selectors, but this time using the labels associated with the node itself - for instance in the `Node` object in Kubernetes.

Caution is strongly advised when applying **host firewall policies**[5], as it is possible to apply a policy that prevents access to the node entirely. In the worst case, it may become impossible to subsequently update host firewall policies for the node or access the node at all.

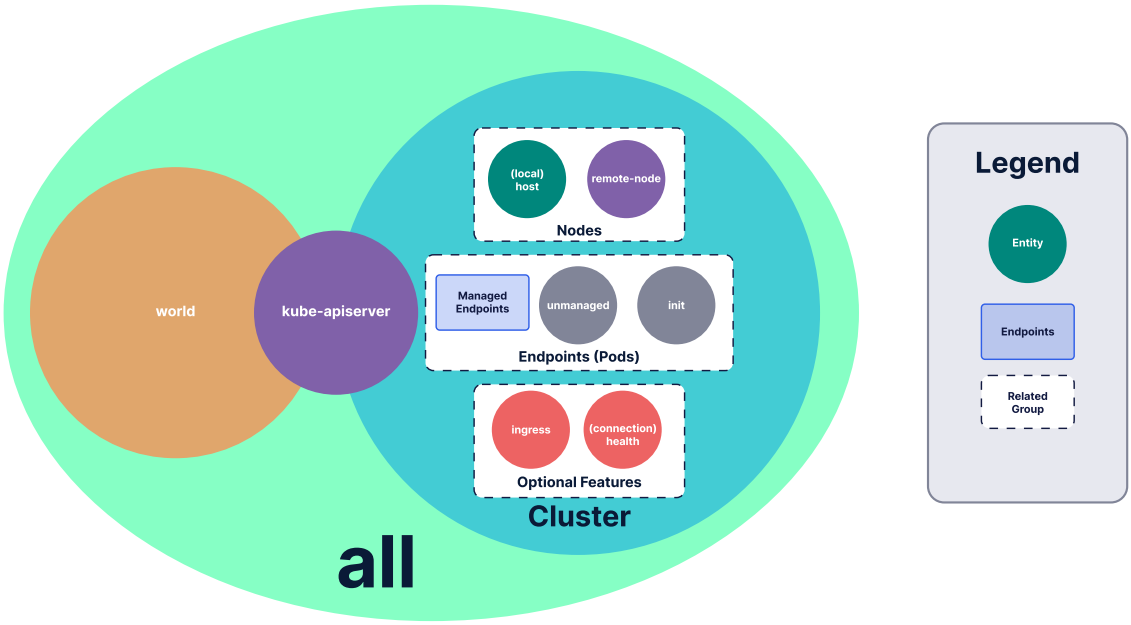
Peer Selectors

The other primary fields under the Cilium Network Policy are the `spec.ingress` and `spec.egress` sections, which contain a list of rules describing who the subject can communicate with. Each rule contains one or more **Peer Selectors** which determine the set of peer identities to which the policy applies. Internally, Cilium represents all policy peers as security identities with defined labels, even when the peer itself is not managed by Cilium. Selectors that determine the target peer of the traffic come in a range of forms. For Cilium-managed Endpoints and Nodes, the form of these selectors matches the subject selectors described earlier. It is possible to allow traffic to or from other nodes based on labels associated with the node, but this may require additional configuration[6]. There are further special considerations for IP Prefixes, FQDNs, Groups and Services which will be covered in more detail in following sections.

[6] **Node-based label selection**[7] was introduced in Cilium v1.16 and requires the `nodeSelectorLabels: true` option to be specified in the Helm configuration. See the **feature status matrix**[8] for more details on feature support.

Entities

As a convenience mechanism Cilium provides default selectors for sets of peers, known as Entities. A `toEntities` or `fromEntities` selector may select communications with all peers in that Entity. Many of the Entities are derived from *reserved labels* as described in **Sources of Label Information**. While the reserved label is a specific instance of a label associated with a network endpoint, the corresponding Entity is a selector that will apply policy for that network endpoint. For instance, creating a `world` Entity selector would match all locations that Cilium determines to be outside the cluster. Conversely, creating a `cluster` Entity selector selects locations *within* the cluster, including the control plane, Cilium-managed Endpoints, and Nodes. A full set of these entities is described below. The `world` Entity can be further customized and subdivided, so it is covered in further detail under **World Selectors**.



This venn diagram represents the overlap between different entities in Cilium. All network endpoints are considered part of the `all` entity, which is primarily composed of two groups of endpoints: Those belonging to the world, and those belonging to the `cluster`. The `kube-apiserver` entity is special in that it may be part of the `world` or the `cluster`, depending on exactly how it is deployed. Within the cluster entity there are six distinct entities: the local host or remote nodes; endpoints that may be unmanaged or initializing; and finally entities for optional features - Cilium Ingress and cluster-wide network health detection.

Entities Selecting Endpoints Inside or Outside the Cluster

The `all` Entity provides an efficient way to select all traffic, whether within the cluster or outside the cluster. This can be a useful construct when your threat model is primarily focused on protecting traffic flowing in one direction, but you allow all connections in the opposite direction. For instance, this can be used to allow all traffic into the cluster as part of a broader policy that primarily limits traffic flowing from inside the cluster towards the outside.

The `kube-apiserver` Entity represents the Kubernetes control plane, identified by traffic corresponding to the control plane's IP address. Depending on how the Kubernetes control plane is deployed, the kube-apiserver entity may be backed by locations inside or outside the cluster. Use of this Entity as part of Egress policy is the most common pattern, and should work in most situations. However, use of this Entity as part of an ingress policy towards Endpoints may be unreliable in some circumstances, as Cilium relies upon the network infrastructure preserving the original control plane IP addresses for such traffic in order to match the traffic. In some managed Kubernetes environments, traffic from the kube-apiserver towards workloads may not preserve the original control plane IP address, and therefore **may not be identified as from the kube-apiserver**[9] entity.

Entities Selecting Endpoints Inside the Cluster

The `cluster` Entity selector selects all peers that logically reside within the current cluster, across all Kubernetes namespaces as well as the underlying infrastructure such as nodes within the cluster, and for any traffic to/from Cilium-managed components such as Kubernetes Ingress or cluster connectivity health checker instances.

All Nodes in the cluster are considered part of the `cluster` Entity, and they are alternatively selectable by either the `host` Entity or the `remote-node` Entity. By default in a Kubernetes environment, all traffic from the local `host` towards a workload on the same node is allowed, in order to facilitate application health checks[10]. In some configurations, traffic from outside the cluster can masquerade as traffic coming from a remote node, so caution is advised when using the `remote-node` entity in network policies. This is further explored in the section **Network Address Translation**.

All Endpoints in the cluster are considered to be part of the `cluster` Entity, whether they are managed by Cilium or not. In a properly configured Cilium installation in a Kubernetes environment, all Endpoints should appear as Cilium-managed endpoints, and the `unmanaged` and `init` entities do not need to be used. There is no specific Entity for all Cilium-managed Endpoints, as this selection can be expressed using an empty Endpoint Selector, "`{}`".

[10] This **policy enforcement mode**[11] behavior is optionally configurable.

There are two additional Entities that select traffic for specific Cilium components, which are both also subject to the `cluster` Entity. The `ingress` Entity selects traffic from all Cilium Ingress instances in the environment. The `health` Entity relates to the Cilium connectivity health (cilium-health) feature, which establishes a periodic ping between all nodes in the cluster in order to detect network disruptions in the fabric. By default there is no network policy applying to the cilium-health components or the nodes such that they would inhibit this feature. However, if a more strict default network policy posture is applied, or Host Firewall policies are created, then this selector can be used in order to explicitly allow connectivity checker traffic.

Selecting External Peers

The `world` entity occupies a special space: In a sense, it represents the least privileged set of possible peers for a network policy. These peers are the accessible network locations that Cilium has minimal information about by default, as they exist outside of the `cluster`. The `world` Entity can be useful by itself in network policy if your threat model defines broad security domains for inside and outside your environment, and you wish to allow all outbound traffic while limiting inbound. However, given that it represents communication with potentially arbitrary Internet-connected devices, it can be useful to divide the World into more specific subsets that represent groups of higher and lower trust locations. You can then manage the policy for communication with these subsets independently.

Attempting to use the following selectors to target traffic within the Cilium-managed cluster will not cause the intended traffic to be selected, and such statements can introduce additional adverse policy processing in Cilium's network policy engine. As such, do not use CIDR, FQDN, Group or Services based policy selectors when the peers reside within your environment.

IP Prefixes (CIDR)

You can define subsets of the `world` entity using their network addresses. CIDR selectors apply policy for peers that fall within the specified IP address prefixes. You can also optionally carve out excluded CIDRs from the selector, for instance if a policy should apply to a wider prefix but exclude a specific range within that wider prefix. Both IPv4 and IPv6 prefixes are supported, providing that Cilium is configured in dual-stack mode.

If you have a set of IP addresses that belong to a common group and you wish to manage this group of IP or IP prefixes collectively, you can create `CiliumCIDRGroup` resources with the list of IPs and then create a CIDRSet selector that references the group by name.

By creating a dedicated resource for the CIDR Group, this also makes it easier to share a common group of IP addresses as your infrastructure changes, and update the list of IPs without modifying individual network policies.

CIDR selectors never select Cilium-managed endpoints, even if an endpoint's IP address is within a given CIDR. Use an EndpointSelector instead. CIDR selectors also do not select nodes within the cluster by default[12].

Fully Qualified Domain Names (FQDNs)

Given that environments are increasingly dynamic, and IP addresses may be reallocated for different purposes regularly, you may find it useful to reference peers by meaningful names. If you use DNS for service discovery, Cilium supports selecting peers by `matchName` or `matchPattern` for the peer's domain name. DNS-based policy relies on Cilium learning DNS information out-of-band from the network policy by proxying DNS messages or by polling for a predetermined set of FQDNs[14]. For a proxy-based deployment, you must explicitly configure Network Policy to **intercept DNS traffic**[16] in order to enforce `toFQDNs` statements. Like CIDR selectors, FQDN selectors do not select Endpoints, Nodes, or Services within the cluster by default[10], and can only define peers outside of the Cilium-managed cluster.

Groups and Services

Cilium also supports syntax for selectors with `toGroups` statements that can match on AWS metadata for allowing traffic towards cluster-external destinations in a particular **AWS Security Group**[17]. Additionally you can allow **traffic to services**[18] without selectors, for cases where you have created a Kubernetes service and manually update the backends to point towards locations outside of the Kubernetes cluster. As these selectors only apply to traffic that is otherwise already considered part of the `world` Entity, these should be used with caution[19].

Application Ports

Much of this chapter is focused on the definition of selectors which provide ways to categorize network endpoints into groups based on metadata known about those groups, such as individual labels or domain names. These selectors can be considered to fall within the domain of Layer 3 in the OSI networking model, as providing network policy language to express how IP traffic should be handled. Cilium also supports selecting traffic based on Layer 4 properties such as the destination port and protocol used for a connection. A rule can allow connections to multiple ports in a single rule[21].

[12] **CIDR-based selection of cluster nodes**[13] was introduced in Cilium v1.15 and requires the `policyCIDRMatchMode: nodes` option to be specified in the Helm configuration. See the feature status matrix for more details on feature support.
[14] As of 2024 (Cilium v1.16), DNS polling is supported only for **Ingress FQDN policies**[15] with Isovalent Enterprise for Cilium.
[19] As of 2024 (Cilium v1.16), extending these to select in-cluster destinations is not on the roadmap. See the **feature status matrix**[20] for more details on feature support.
[21] As of 2024 (Cilium v1.16), Cilium supports specifying arbitrary port ranges in a port rule.

Layer 4 matches can be specified either as part of a rule with a Layer 3 sector, or in a dedicated rule. If specified in policy at the same level as the selectors described above the rule allows the subject to communicate with a destination only if the traffic is directed to a specific port/protocol. If the Layer 4 properties are specified alone in a rule, this allows traffic towards a particular port regardless of the Layer 3 destination. A Layer 4 policy rule alone will not enable ICMP packets to reach a Layer 3 destination.

Policy rules with an application port can also optionally specify a specific expected set of rules at Layer 7 in order to provide specific application-layer network policy enforcement. These rules instruct Cilium to parse the corresponding traffic as the target protocol, such as DNS or HTTP, and drop traffic that does not conform to that protocol. Furthermore, protocol-specific rules can be expressed, such as to allow DNS requests for specific domains or to allow particular HTTP queries to be made. Traffic that does not match the rule is denied using native protocol rejection messages such as HTTP 403 “Forbidden”. For more details about usage and support, see the **feature status matrix**[22].

Policy Actions

The final primary component of a network policy is the action which Cilium enforces for the selected traffic. Cilium supports allowing or dropping traffic that is either leaving (egress) or arriving (ingress) at a network enforcement point. These are expressed at the top level of a network policy in the form of an `ingress`, `egress`, `ingressDeny`, or `egressDeny` statement with a list of peer selectors nested underneath. Some more advanced Cilium features may apply additional constraints to traffic affected by the policy, for instance to require that endpoints establish mutual authentication, allow traffic based on SNI, or to terminate TLS connections. These more advanced features are typically specified as parameters within an application port rule.

The Cilium Network Policy Model

The use of the Security Identity as a primitive and the use of selectors to apply policies provides a critical basis for understanding the way that Cilium operates. However, an understanding beyond these base concepts and capabilities can help with operating Cilium Network Policies. This section applies those concepts to the Cilium architecture to describe the Network Policy Model that Cilium implements. This section starts from a high level, considering which components apply network policies and how, and then progresses deeper into the implementation to describe the way that Cilium's eBPF dataplane enforces these network policies right down to the per-packet level.

This section does not intend to comprehensively cover all possible features and configurations in a Cilium environment, but it does reference many of the commonly deployed features, and should provide a basis for how to reason about additional features. Topics not covered in detail here can be explored through content and hands-on labs available at **docs.isovalent.com**.

Roles in a Multi-Tenant Environment

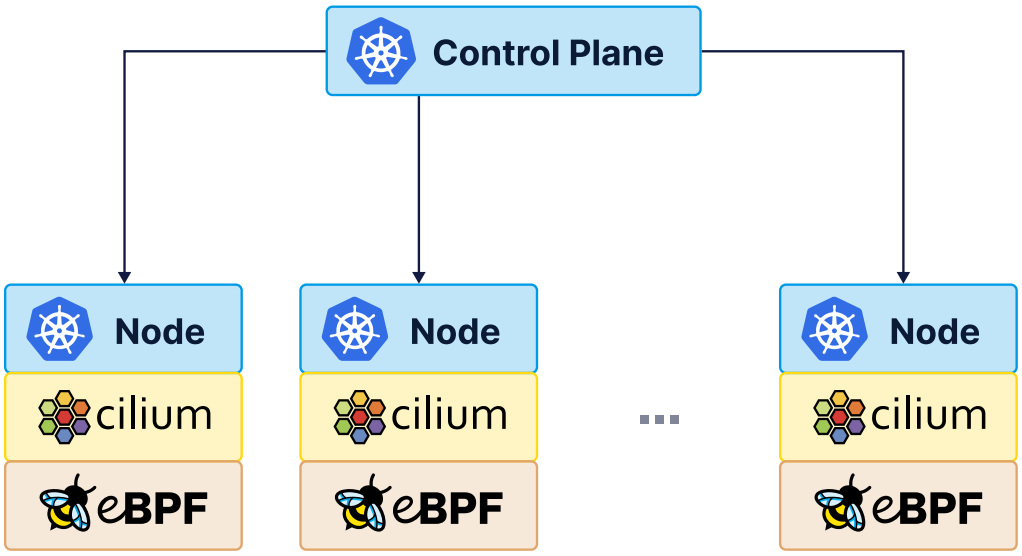
Modern cloud-native environments are often managed by multiple teams. A common scenario is that there is one team responsible for the overall platform security who implement cluster-wide or organization-wide network policy controls, then additionally one or more application teams who are responsible for a portion of the cluster. In a Kubernetes environment, these roles are supported by role-based access controls that determine the set of resources that can be managed by individual teams. Cluster administrators may have access to the entire cluster, then delegate control over specific namespaces to other teams.

In order to facilitate this shared management of network policies, Cilium supports multiple separate network policy resources. *CiliumClusterwideNetworkPolicy (CCNP)* expresses policies that may apply across an entire cluster, and has constructs to target the infrastructure itself for *host firewalling,* as well as constructs that define policies across a range of applications in individual namespaces. In addition to CCNP, Cilium also supports per-namespace network policies which are intended for use by individual application teams. For namespaced network policies, Cilium supports standard Kubernetes NetworkPolicy as well as CiliumNetworkPolicy.

If your organization is multi-tenant and distributes the ownership over network policy across multiple teams, then good practice is to configure role-based access control to grant write privileges for CCNP resources to the team responsible for platform security and restrict write access for other teams. Access to namespaced network policies can then be delegated to individual teams, subject to your internal threat model.

Applying Network Policies

Cilium receives the intent of user-defined Network Policy primarily from the Kubernetes control plane in the form of Cilium Network Policy (CNP), Cilium Clusterwide Network Policy (CCNP), and Kubernetes Network Policy (KNP) resources. Both CNP and CCNP follow almost identical syntax, with the primary difference being the namespacing of these resources in a Kubernetes context. The first step to applying a network policy is triggered when you apply the policy into the cluster. This triggers the control plane to parse and validate the network policy, and store it as desired state for the cluster to process.



Cilium operates in a model with a centralized control plane that informs each node about the relevant resources that Cilium must ingest and implement, which Cilium achieves by programming the eBPF dataplane on that node.

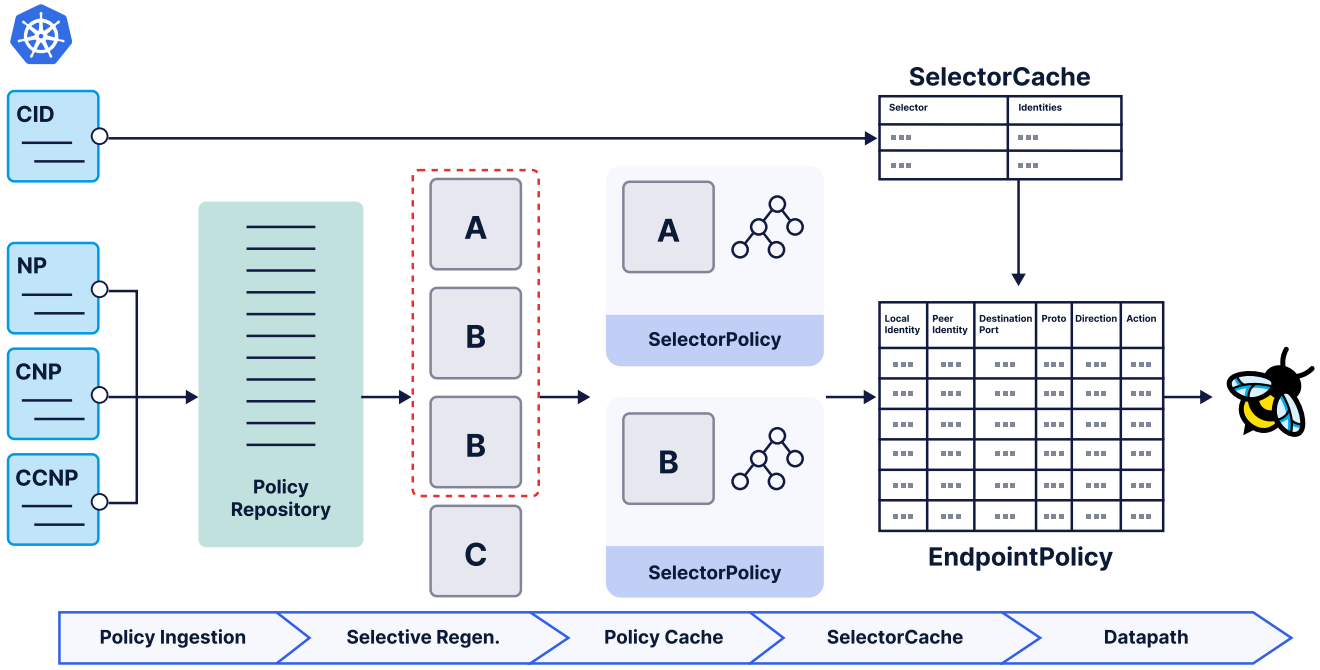
The cilium-agent component on each node within a cluster independently pulls all of the information necessary to implement network policies for the local node from the central control plane. In practice, this includes not only the network policy resources, but also other resources that inform Cilium about how to correlate traffic with the labels used in network policies. Examples of this are standard Kubernetes Pod and Node resources, CiliumEndpoints (CEP), CiliumEndpointSlices (CES), **IsovalentFQDNGroup**[23] and **CiliumCIDRGroup**[24].

Policy Calculation on Each Node

Upon agent startup, or when resources are otherwise synchronized from the central control plane, the Cilium Agent calculates the network policy enforcement for all Endpoints on the node where it is deployed. Logically, you can consider this network policy calculation to go through each locally-present Endpoint, evaluate which network policies apply to that Endpoint, then furthermore evaluate which remote peers should be accessible based on the Selectors in the policies. The result of this computation is a series of configurations for Cilium's eBPF datapath and each enabled proxy, such as the in-built or standalone DNS proxy as well as the Envoy proxy.

With usage patterns in cloud-native environments often involving many deployments, and the change in scale of these deployments frequently changing, the demands on the network policy calculation engine can be quite high. Cilium has been optimized to perform policy calculation in a series of stages, each of which is cached in order to facilitate faster incremental policy calculation based on changes occuring in the cluster. The primary stages are:

- Ingesting all network policies of each supported resource type and assembling into a single internal representation of the policies, known as the *policy repository*;
- Selecting sets of locally-relevant policies from the policy repository to create a tree of policy filters per Security Identity, still operating in terms of selectors - *a selector policy;*
- Preparing a mapping of each individual selector statement from each policy to the set of Security Identities that match the selector - *the selector cache*;
- Distilling the selector policy into a concrete set of rules defining which Security Identities can reach which other Security Identities and on which ports - *the endpoint policy;* and
- Injecting these low-level policy keys and values into the eBPF *policy maps* for enforcement in eBPF on a per-packet basis.



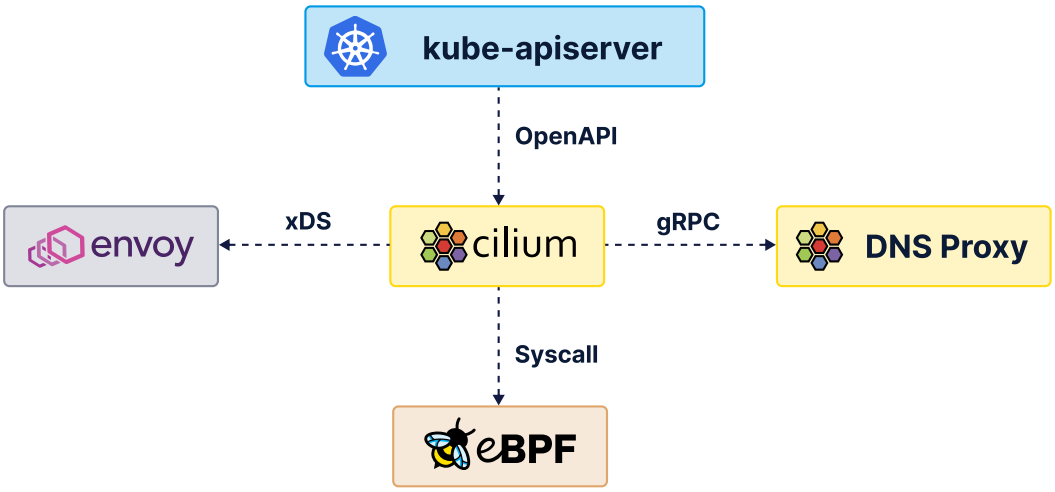
Cilium processes Kubernetes policy objects from left to right towards the eBPF dataplane. Several stages are executed. Cilium ingests the network policies into the Policy Repository, selectively regenerates local Endpoints to create a selector policy for each local Security Identity (A and B in this example). The Selector Cache processes Security Identities and maps them to selectors. The penultimate step evaluates the selectors in the SelectorPolicy using the SelectorCache, turning them into concrete rules referencing specific Security Identities. The resulting EndpointPolicy rules are injected into the eBPF dataplane.

Cilium provides powerful expressions to decide which network policies should apply to which subjects and peers, by including or excluding particular label keys and values. These expressions can create complex interactions between several rules and any particular subject or peer traffic. However, the resulting policy ultimately chooses one action for a particular set of traffic that flows through the system - to allow the traffic, deny it, or redirect the traffic for further inspection at Layer 7. Therefore one of the key responsibilities of the policy calculation engine is to evaluate the priority of rules, disambiguate the intended policy impact on the targeted traffic, and translate those complex statements into a simpler set of match and action classifications that can be executed in a machine-friendly manner. This machine-friendly format is injected into Cilium's per-packet enforcement layer in eBPF for efficient processing.

Layer 7 Functionality

Where possible, Cilium implements its features natively in eBPF in order to keep the traffic inside the operating system kernel for efficiency. For a variety of functionality including network layer 3 / 4 enforcement, multi-cluster, mutual authentication and observability, this is already possible in eBPF directly. However, for some more advanced functionality like Layer 7 traffic management and resilience, TLS termination and Layer 7 policy enforcement, Cilium currently delegates responsibility to userspace proxies to implement the functionality. Cilium supports a range of such features using Envoy, and if those features are enabled—even features not directly related to network policy—the network policy engine is responsible for ensuring that the target traffic reaches the proxy.

Similar to the eBPF calculation, for Layer 7 functionality the policy engine assembles and retains the L7 configuration along with the distilled filters and then configures the individual proxies to apply the desired policy. Depending on the proxy, Cilium may share this configuration using direct function calls, gRPC or xDS protocols. If the policies are shared to an external process, then this configuration is passed through a locally mounted unix domain socket that is shared between the cilium-agent and the corresponding proxy. For DNS traffic, Cilium has a proxy built into the cilium-agent component itself, and Isovalent Enterprise for Cilium also has an option for a standalone DNS proxy which provides high availability during agent restart or upgrade. For other Layer 7 functionality, Cilium delegates the functionality to an Envoy instance running on the same node.



The Cilium Agent on a node pulls network policy configuration objects from the Kubernetes control plane, then calculates the required configuration in Envoy, the Cilium DNS Proxy and eBPF in order to enforce the network policy.

Even when Cilium's Layer 7 functionality is implemented through a userspace proxy, Cilium configures the forwarding through the eBPF dataplane in order to optimize the packet path for delivering packets towards the proxy. All packets flow through eBPF and must pass through eBPF network policy enforcement. Packets subject to Layer 7 functionality are then directed towards the userspace proxy by the eBPF dataplane. The proxy applies the relevant feature which may reject the request or forward it on to another destination.

Datapath Model

The goal for the steps described above is to take a human-friendly representation of policy intent and convert this into a machine-friendly format for efficient policy evaluation at the network layer. That process is specifically tailored to minimize the number of operations required to enforce the policy, thereby minimizing connection latency. At the same time, it is also efficient to handle churn of applications as they are scaled up and down, as the expensive policy calculation step is dependent only on selector statements and security identities, but not on the location that the workloads are deployed (their IP addresses).

These are great properties to have, but these properties can only be provided by an opinionated dataplane that is designed for this policy model. This section describes how the eBPF dataplane is prepared and bound to Endpoints deployed locally, explores the processing pipeline where the policy is enforced, and identifies network factors that affect the way that Cilium determines the security identity of the communications.

Datapath preparation

When a new workload is deployed onto a node, the container runtime issues a Container Network Interface (CNI) “ADD” command to the Cilium CNI on the node. The Cilium CNI binary subsequently makes a REST request over a local unix domain socket to the cilium-agent to create an Endpoint object for the workload. If this is the first workload created on the node, then Cilium goes through a full initialization process of assembling workload metadata, compiling tailored eBPF code to handle packet processing for that Endpoint, loading the resulting artifacts into the kernel, and attaching the eBPF programs to the Endpoints on the corresponding network interface. Finally once all of these processes complete, the CNI ADD command returns a successful result back to the container runtime.

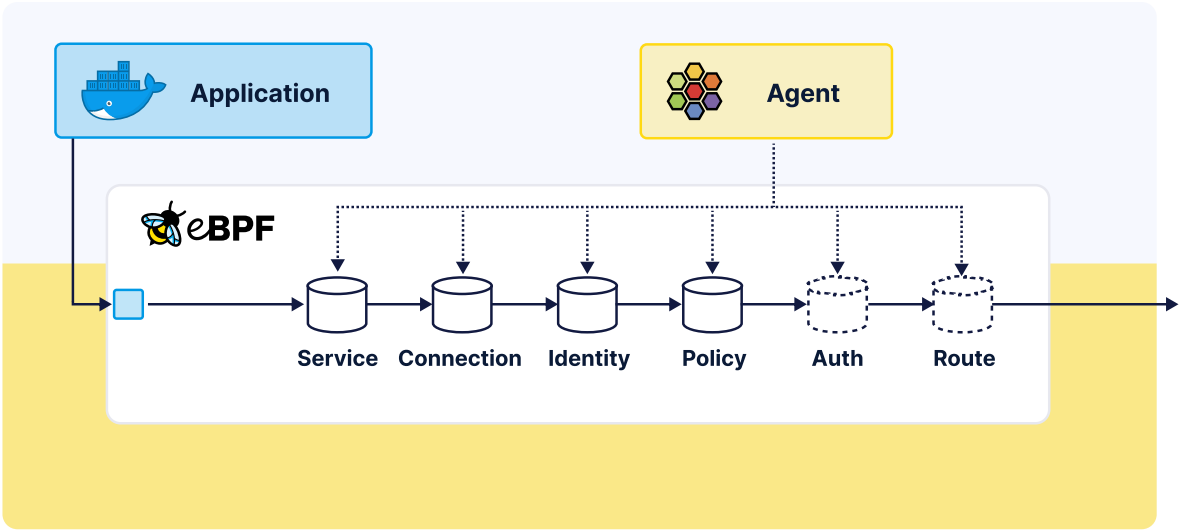
Cilium’s datapath initialization logic, flowing from left to right starting from assembling metadata about a target Endpoint, then compiling the eBPF dataplane for it, loading eBPF objects and attaching them into the kernel, and finally returning the result of these operations back to the container runtime as a CNI ADD result.

Some of the steps of this process are subsequently optimized, for instance the eBPF program compilation steps tailor the intended eBPF functionality primarily using the node-wide agent configuration, so it is often sufficient to compile once and then cache the result of compilation for reuse during subsequent workload deployments. Similarly, it’s possible to share the results of policy computation for map content across multiple instances of the same workload on the same node.

Per-packet operations

In order to deeply understand how Cilium enforces policy for connections initiated in a Cilium-managed network, you need to understand a few things about the structure of the eBPF datapath. While the following description will not comprehensively cover the "life of a packet" in a Cilium environment, it will highlight some key steps in that path which you can use in order to better understand the enforcement decisions that Cilium makes. Cilium's datapath enforcement is primarily applied at the packet layer as a workload transmits packets from the workload's network namespace or VM context towards the underlying host. These packets are transmitted through the host's operating system kernel and reach a "hook" point where the kernel executes the Cilium eBPF programs. The following diagram provides a visual representation of what happens when the packet arrives at the Cilium dataplane.

The Cilium datapath does not represent all possible operations as a list of rules that the kernel can jump between based on the matching of packet content. Rather, it structures the lookup and execution of different networking facilities through a series of feature-specific maps. Each of these maps can be queried using the `cilium-dbg` CLI within the Cilium container, or by capturing a **sysdump**[25] (system dump) from a live environment and inspecting the resulting files.



Cilium agent programs various maps in the eBPF dataplane, represented as cylinders. The flow left to right represents how eBPF processes a packet from an application, with reference to specific maps: progressing from service translation, connection correlation, identity association, policy enforcement, mutual authentication, through to routing to the next hop in the network. The logic for some maps depends on Cilium's configuration, represented with a dashed outline. Mutual Authentication and other Layer 7 functionality may require packets to be forwarded to a userspace proxy for further processing.

Three properties that are useful to understand in order to inspect the active policy enforcement occuring in the Cilium dataplane are that:

- Cilium applies service translation before policy enforcement decisions;
- Cilium defines policy on a per-connection and not per-packet basis, which is implemented through a local cache of live connections—meaning that if a request is allowed by the network policy, then replies are also allowed by the same rule; and
- Cilium determines the security identities of traffic as a dedicated step prior to applying the network policy. This includes determining both the source and destination security identity.

Once Cilium completes these steps for a packet flowing through the system, it enforces the network policy based on the rules generated in the process described earlier. When network policies are updated, the new rules are generated and injected into the eBPF dataplane, which enforces the new policy on existing connections as well as new connections.

Network Address Translation

In a networking dataplane, there is often the need for expressing both real and virtual addresses in order to abstract away the physical location of services and assist with application service discovery. A common application design has the application look up a destination by its DNS name, then the DNS server provides a virtual IP address that is not assigned to a specific location in the network, and the application connects to this address. When the packets leave the workload context, the underlying network is then responsible to translate the virtual IP address into a real address, and load-balance the communications to a backend. In order to provide a consistent experience for users of network policy, Cilium performs this destination NAT (DNAT) step prior to consulting the network policy. That is to say, if an application contacts a virtual IP address, Cilium first translates the traffic to its real corresponding IP address, then determines the security identity of that translated network endpoint, and enforces the policy based on this security identity (in that order).

Another case that may impact network policy enforcement is the use of source NAT (SNAT). Particularly notable is when Cilium implements Kubernetes NodePorts, where a specific port is exposed on all nodes in the cluster. When traffic arrives at a node port, the traffic may be subject to both DNAT and SNAT in order to ensure that subsequent traffic for the same connection is routed to the same backend, and the replies are symmetrically routed back to the client. In this case, the traffic routed through the initial node will adopt the source address of the node itself, so it can become difficult to understand the true origin of the traffic - does it come from outside the cluster via a NodePort, or does it come directly from an application running on that node?

Depending on your environment constraints and threat model, Cilium provides different solutions to handling network policy when network address translation occurs, including **client source IP preservation**[26] or retaining the Security Identity of SNATed traffic **via tunnel headers**[27]. In native-routing mode, such traffic is classified as having the `remote-node` security identity despite the traffic originating outside the cluster.

Treatment of Locally Assigned Addresses

Traffic that originates from outside the current node which undergoes SNAT and then is delivered to a workload on the local node is not classified as from the `host`, even if the IP address indicates that the traffic originates locally. Rather than trusting that traffic is from the local node based on IP addresses, Cilium uses eBPF to classify traffic coming from sockets opened on the same node in order to identify that traffic as coming from the local host - meaning that only traffic originating from local applications on the host are considered as coming from the host itself. Typically, traffic from a local address that is not from a locally running application will appear as coming from the `world` instead.

In general, in a well-configured environment it's expected that traffic between workloads within the same cluster will either be directly routed via "native routing" if the underlying network supports routing the workload IP addressing scheme, or routed via a tunnel when Cilium is deployed in tunneling mode. In the latter case, all traffic between workloads in the cluster is encapsulated - meaning that the full network packets include two sets of IP headers. The outer headers contain node IP addresses, indicating the packet is directed from node IPs towards node IPs. The full original packets between workloads are retained inside these packets through inner headers and packet data. Cilium should not be configured to masquerade workload-to-workload traffic using SNAT to Node IPs, as this may lead to Cilium treating traffic to or from workloads inside the cluster as being sent/received from remote nodes.

Each Cilium agent in the cluster upon startup will automatically detect IP addresses that are assigned to local interfaces, and notify other agents on other nodes that these IP addresses are assigned to that node. This allows other nodes within the cluster to classify traffic and apply policy with those IP addresses as traffic to or from the node, rather than classifying that traffic as to or from the `world`. The full set of such IP addresses can be inspected by looking at the CiliumNode objects shared through the control plane, or by locally querying the cilium agent using the `cilium-dbg` command-line inside of a Cilium container.

Enforcing the network policy

In order to enforce network policy in the dataplane, Cilium must determine the security identity of the source and destination of the traffic. The specific mechanism used for determining these security identities varies depending on the enforcement point and the configuration modes of Cilium, such as whether the routing mode is configured for native routing or tunnel. For egress traffic from an endpoint, Cilium inherently knows the source identity of the traffic, as Cilium is directly attached to the network device that emits the traffic; Cilium embeds the source security identity into the eBPF programs. Similarly, for traffic that ingresses towards a workload through its network device, Cilium embeds the destination security identity into the eBPF program. In these cases, Cilium does not need to perform any lookups to determine the security identity association.

Another case where Cilium does not need to perform lookups is when Cilium is deployed with the routing mode set to tunnel when handling ingress network policy. In this case, Cilium running on the source node inherently knows the security identity of the workload, and transmits this security identity along with the packet as it is routed towards the destination in the cluster. This typically uses a VXLAN or Geneve virtual network identifier field. Then, when the packet arrives at the destination node, Cilium routes the packet towards the destination's eBPF program which then uses the source security identity bound to the packet and the destination security identity bound to the workload network device and then performs the policy lookup to apply the network policy.

Two other cases are commonly handled via a distributed IP to Identity cache (IP Cache): for the peer identity for egress (all modes) and ingress (native routing only) cases. When you deploy a workload onto a Cilium-managed node, Cilium centrally tracks the corresponding workload's IP address and corresponding Security Identity, and distributes this mapping to each node in the cluster. This prepares the node for connections that are subsequently established by the applications in the environment. So, when a workload initiates a connection, Cilium performs a lookup for the remote peer's security identity in this IP cache and then uses this security identity to enforce the policy. This security identity lookup can be further enhanced with **mutual authentication properties**[28].

Once the security identities of the source and destination are known, Cilium has performed service translation, correlated the packet with the original connection, and identified the network port of the traffic, Cilium can perform a lookup in the *policy map* which was previously populated according to **Policy Calculation on Each Node.** The result will either allow the traffic, deny the traffic, or subject the traffic to further inspection in userspace, according to the intent outlined in the network policy. The next chapter will explore in more detail which capabilities are available in Cilium for these enforcement actions.

Operating Cilium with Network Policies

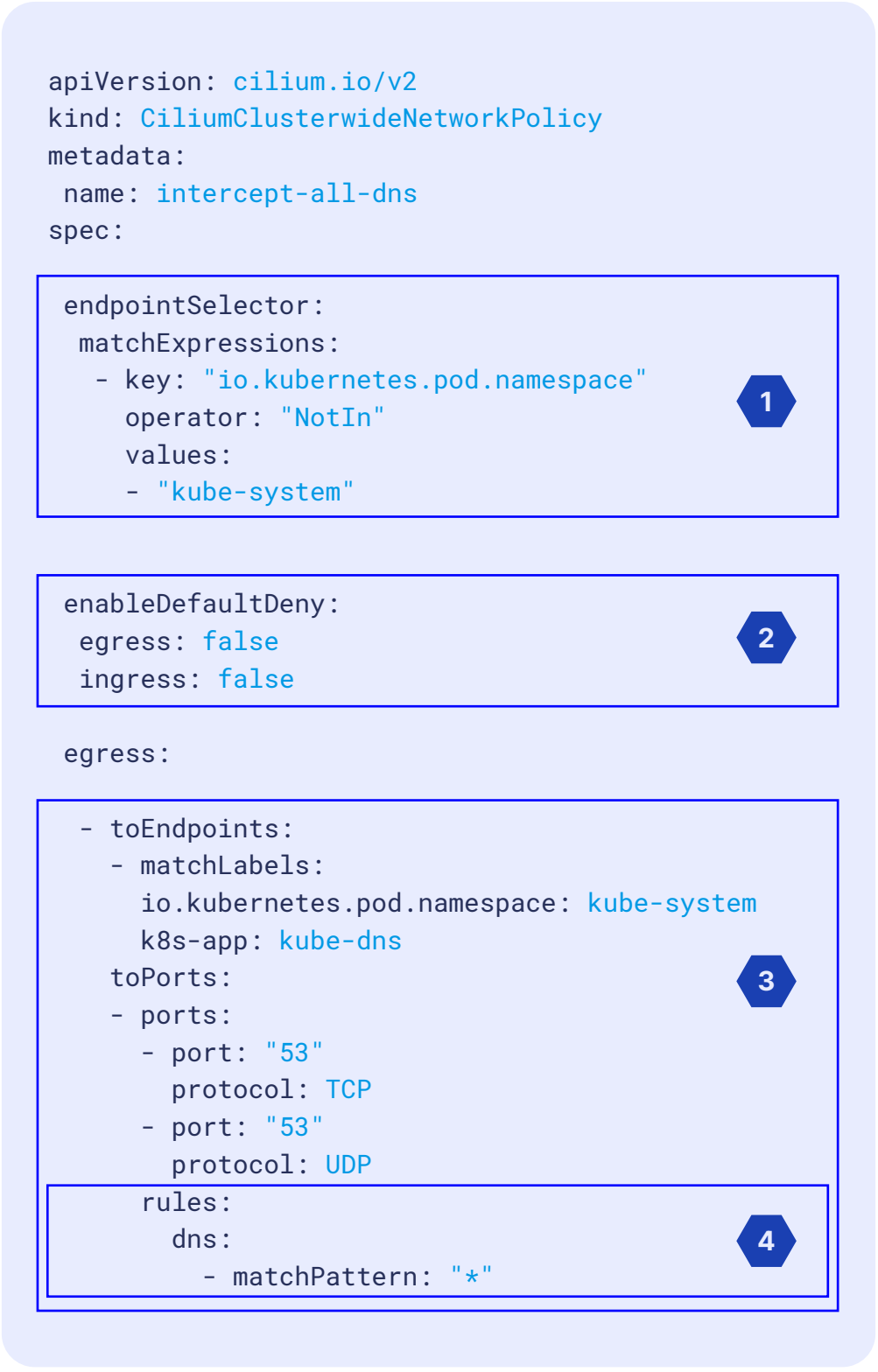
At this point you should have a solid grasp of the core network policy model in Cilium from a high level. This chapter intends to expand on these concepts from a more practical perspective, describing how to write network policies, troubleshooting live environments, and understanding the way that the network policies affect your environment at scale. This will cover how to monitor and observe your environment, but it will not comprehensively break down each network policy use case or way that all Cilium features interact with the network policy engine. Rather, this chapter should provide insight into common usage patterns and give you the tools to be able to further explore Cilium's capabilities through production usage.

Writing Network Policies

Cilium's network policy engine provides a range of capabilities through the KNP, CNP and CCNP APIs, which are typically written in YAML and managed through a central control plane in the cluster. This section provides an overview of the structure and semantics of these policies, and discusses the way that these rules may interact with external systems. For more specific examples to write network policies for your environment, see the **Isovalent labs for security**[29].

Key Components of a Rule

Network policy identifies the peers involved in network communication and how the policy intends for that communication to be managed - for instance, to allow or deny the traffic. In practice, Cilium identifies four key areas that reflect this intent in order to implement the network policies: the enforcement point, defined by the subject selector; the default deny posture for the subject of the policy; a set of rules under an ingress or egress stanza which identify specific traffic that should be allowed or denied; and finally, an optional rules statement that subjects the traffic to deeper inspection. The following figure highlights these four important components of a common network policy:



An example Cilium Network Policy with four sections highlighted: (1) the subject selector; (2) the default deny posture of the policy; (3) An individual rule including peer and port selectors, which is under an egress section; and (4) the layer 7 component of the rule. When any workload outside the kube-system namespace attempts to make queries to kube-dns on port 53, that traffic will be allowed by this policy and Cilium will track / cache DNS information for use in observability and enforcement.

Policy Subjects

This example policy above is a `CiliumClusterwideNetworkPolicy` applying to multiple namespaces, and it uses a `matchExpressions` statement to identify the subjects of the policy where this network policy will be enforced. This statement selects workloads based on the Identity-Relevant Labels, which typically includes most container labels but excludes some common labels with high entropy, such as UUIDs. The section (1) selects all Cilium-managed workloads that reside in namespaces other than the "kube-system" namespace. This statement will not select any workloads in those namespaces with `hostNetwork: true`.

If you create a KNP or CNP resource, then the subject selector implicitly selects the namespace that the resource is created in. It's not possible for a KNP or CNP to apply policies for endpoints outside the namespace where the network policy is created.

Default Deny Posture

It's possible to express whether the network policy should enable default deny for the subjects by using an `enableDefaultDeny` stanza. This example policy does not enable default deny due to the explicit configuration under section (2). Using a form like this can be useful for clusterwide policies, as the policy applies rules clusterwide without forcing teams to introduce network policies for each workload in the cluster to define which other connections should be allowed.

If the section (2) was missing then the network policy would put the subject endpoints into a default deny posture. Specifically, if there is an `egress` or `egressDeny` stanza inside the spec with at least one rule, then the workload by default will be put into a default-deny security posture for traffic leaving the workloads. Similarly, if there is an `ingress` or `ingressDeny` stanza inside the spec with at least one rule, then the workload would be put into default-deny for incoming connections. Overall, if there is at least one policy configured somewhere across KNP, CNP, CCNP that selects a workload and has a rule within one of these stanzas, then the endpoint will be configured to deny traffic implicitly by default for that direction. The rules within these stanzas then explicitly allow or deny traffic on top of this implicit default deny.

Peer Traffic and Policy Verdicts

A valid network policy has at least one rule inside of an `egress`, `egressDeny`, `ingress` or `ingressDeny` stanza which describes a set of traffic that should be allowed or denied. This example `egress` stanza defines a set of allowed traffic, in this case a single rule indicated in (3) which itself has multiple matching constraints. The rule matches workloads in the cluster with the kube-system namespace and k8s-app label set to kube-dns - the DNS server commonly deployed in a Kubernetes cluster. Additionally, the rule has a constraint that only traffic towards kube-dns on port 53 should be allowed, specifically if using TCP or UDP.

This example is a CCNP and specifies the namespace of the peer. CCNP resources that do not specify the namespace may match peers in any namespace by default. However, in the case of KNP or CNP resources, if there is a peer selector which does not specify the namespace, then the endpoint selector implicitly only allows traffic with the specified labels within the same namespace as the KNP or CNP resource.

Finally, this rule also has a ports `rules` stanza as highlighted in (4). In this case the port rule instructs Cilium to treat traffic on this connection as DNS, and the DNS traffic should match the pattern "*" (wildcard all) for queries. Due to the pattern, Cilium will inspect the DNS traffic and allow all DNS queries and responses. Cilium uses this `rules` stanza as a hint that the traffic needs deeper inspection, so Cilium redirects this traffic through a userspace proxy in order to implement additional Layer 7 functionality.

Rule Priority

You may be familiar with existing firewall languages which provide an explicit priority parameter. As of 2024, Cilium does not provide an explicit priority configuration in the policy language[30]. Instead, Cilium treats all rules as belonging to one of two specific categories, based on the final verdict expected for this traffic: A rule either expresses that traffic should be allowed or denied. You can think of the logic for rule priority in Cilium to follow the following three questions in order:

1. Is there any explicit rule to deny this traffic? If yes, drop it.
2. Is there any explicit rule to allow this traffic? If yes, allow it.
3. Is there any policy enforcing a default deny posture for the subject? If yes, drop the traffic. Otherwise, allow the traffic.

If a rule exists within an `egressDeny` or `ingressDeny` stanza, then that rule instructs Cilium to always drop the specified traffic. This traffic restriction applies even if there is another Allow rule selecting the same traffic in an `egress` or `ingress` stanza. This makes Deny rules suitable for enforcing policies where workloads must not connect to a particular destination under any circumstances. When used in a CCNP with the relevant role-based access control, this can enforce a drop policy that cannot be overridden by namespace owners.

Language

The Cilium network policy language is commonly represented in YAML, with different YAML constructs defining the semantics around the rules that apply to network traffic. For instance, collections are used to define the overall set of rules for a particular direction such as ingress or egress. Furthermore, the fields in each element of a collection define a group of common parameters that must match in order for the rule to apply to traffic. This section provides a primer on some of the common patterns in this YAML format with examples in order to demonstrate how Cilium applies network enforcement based on the rule content.

[30] Rule priority is on the roadmap for addition to Cilium Network Policy in Isovalent Enterprise for Cilium.

Consider the following example rules:

```
apiVersion: cilium.io/v2
kind: CiliumNetworkPolicy
metadata:
 name: allow-dns-lookups-cilium-io
spec:
 ...

 egress:
   - toPorts:
     - ports:
       - port: "53"
         protocol: TCP
```

1

```
   - toEndpoints:
     - matchLabels:
         k8s-app: kube-dns
```

2

```
   - toEndpoints:
     - matchLabels:
         io.kubernetes.pod.namespace: kube-system
         k8s-app: kube-dns
     toPorts:
     - ports:
       - port: "53"
         protocol: TCP
       rules:
         dns:
           - matchPattern: "*.cilium.io"
```

3

An example Cilium Network Policy with three individual rules highlighted, each of which allows a subset of connections egressing the subject: (1) a rule selecting all traffic egressing on port 53/TCP; (2) an overlapping rule matching all traffic egressing towards an endpoint with the label “k8s-app: kube-dns”; and (3) a rule that selects traffic towards endpoints with the kube-dns labels and kube-system namespace, which must also match port 53/TCP, and also must treat that traffic as DNS and match a wildcard pattern on the DNS content.

Conjunctions

The `egress` stanza of the example policy represents a list of unordered rules, each of which is delimited by an indented dash. There are three such items in the collection, each representing an egress rule that defines a set of traffic that is allowed by the policy. You may read the policy as stating that the subject endpoint may initiate egress connections to the traffic specified in either rule (1), (2), or (3), and any traffic matching any of these rules will be allowed[31].

Consider the rule highlighted by the box (1). The single egress rule has a `toPorts` field which itself is also a list of port selectors. In this case there is a single port selector which itself has two properties each which must match in order to successfully select the traffic: it selects port 53 and the TCP protocol. This defines that the subject endpoint is allowed to initiate connections outbound towards any location on that port and protocol. The lack of a `toEndpoints`, `toCIDR` or other modifier within the rule means that this rule effectively "wildcards" those parameters, meaning that the rule matches all traffic on port 53/TCP, regardless of who the connection is initiated with. However, if an outbound connection was on port 53 using UDP, the traffic would not be allowed by this rule. Furthermore, if the outbound traffic used a different port number with TCP, the traffic would also not be allowed by this rule.

Composability

Rule (2) contains an endpoint selector selecting endpoints with the label "k8s-app: kube-dns". This allows all connections towards endpoints with that label, regardless of port or protocol. When put in context of the policy example above, there is an overlap with rule (1) specifically for connections towards endpoints with the labels "k8s-app: kube-dns" on port 53/TCP. While that traffic was already allowed by another rule, this rule also allows connections on other ports and protocols. Given that this rule already expresses that all traffic towards endpoints with the labels "k8s-app: kube-dns" should be allowed, the only way to then restrict traffic towards that endpoint in conflict with this rule would be to create an additional overlapping rule in an `egressDeny` statement. Note that it is possible for such rules to overlap, even if the conflicting rule matches on different labels. Consider a workload with the labels "k8s-app: kube-dns" and "protect: true". The policy example above would select this endpoint with the peer selector. If you created another network policy with an `egressDeny` rule that matched the labels "protect: true", then the traffic to that endpoint would be denied. This is because both policies apply to traffic towards the endpoint, and the **Rule Priority** defines that the Deny rule takes precedence over each allow rule, even if both rules select the same peer. The Debugging section goes into more detail about how to inspect and identify such conflicts by **Monitoring live traffic with Hubble.**

Nested Statements

The rule (3) in the example above has a series of nested properties which further modify the scope of the rule. The rule itself is defined by two properties: There is a `toEndpoints` statement and a `toPorts` statement. For an egress connection to be allowed by this rule, both parameters must match. This is a different form from rules (1) or (2) which each select only one of these parameters.

[31] Unless otherwise overridden by a Deny policy as described under **Rule Priority**.

The endpoint selector further narrows the scope to endpoints which only have both the namespace kube-system and the k8s-app label kube-dns. The port rule as the first entry in the list under `toPorts` itself is also interesting: It specifies two properties that must match in order for this rule to allow traffic. At least one of the `ports` statements must match, but also the `rules` statement must match traffic for the traffic to be allowed.

The rules statement treats the traffic as DNS, and allows DNS queries and responses that match the pattern "*.cilium.io". For this rule to allow any traffic, it must match all of these properties. You can read the entire rule (3) as follows: The subject may make requests towards endpoints in the kube-system namespace with the k8s-app label set to kube-dns, if the connection is towards port 53 using TCP as the layer 4 protocol, DNS as the layer 7 protocol, and the connection is making a request for direct subdomains of cilium.io.

While rule (3) only allows a subset of traffic towards port 53, this rule actually overlaps with rule (1) which is more permissive. Given that rule (1) allows all connections on port 53/TCP, the resulting behavior of the overall policy with all rules is that Cilium allows all DNS requests on this port/protocol. That is to say, the nested "rules" statement in rule (3) does not apply a restriction to rule (1) for the match pattern. Rule (1) still allows all requests on that port and protocol, so therefore requests for other DNS domains are also allowed.

There is one further nuance about these rules: Since rule (3) creates the expectation that traffic towards kube-dns or port 53/TCP is DNS traffic, all L3/L4 requests matching these parameters will be sent to the DNS proxy for inspection and observability, whether they match the pattern "*.cilium.io" or not.

Layer 7 Filtering

The examples in this section make use of the `rules` statement as a property of the port rules, using DNS properties. This kind of statement can alternatively match on HTTP, gRPC and other layer 7 protocols[32]. Furthermore, there are other supported properties under the port rule that may not act as a match on the traffic, but rather apply other functionality such as providing TLS connection termination or bridging the traffic with custom Envoy filters. For more network policy features, see the **Isovalent Enterprise for Cilium documentation**[34].

Integrations with External Components

Cilium integrates with other connected systems such as the Kubernetes control plane, the Domain Name System, and optionally other clusters. This section provides some context about these interactions that you may find useful while operating Cilium clusters in production.

[32] The Isovalent Enterprise for Cilium documentation lists **supported Layer 7 protocols**[33].

Compatibility with Kubernetes Network Policy

Cilium's own network policy resources were heavily inspired by Kubernetes Network Policies, and the semantics should be largely the same, but for an accurate definition of the intent of KNPs, see the **upstream Kubernetes documentation**[35]. Cilium network policies support a superset of functionality in standard KNP, so when Cilium imports KNP, it first converts the KNP into the equivalent internal format expressed in the Cilium Network Policies, then subsequently calculates how this policy impacts the local node.

This design provides one notable impact to Cilium's interpretation of Kubernetes Network Policy: Cilium categorizes KNP `ipBlock` statements as part of the world Entity, meaning that these policy statements in KNP also only apply to peers not managed by Cilium.

DNS-based Policies

Cilium has two mechanisms for learning DNS information to inform the policy engine how to apply policies based on domain names: For egress FQDN policy, Cilium supports in-line DNS interception via a DNS proxy. For ingress FQDN policy, Isovalent Enterprise for Cilium supports polling FQDNs. In each of these cases, Cilium relies on the DNS records present in these DNS servers, and makes network policy decisions based on this information.

```
apiVersion: cilium.io/v2
kind: CiliumNetworkPolicy
metadata:
  name: allow-cilium-io
spec:
 endpointSelector:
  ...
 egress:
  - toEndpoints:
     - matchLabels:
        io.kubernetes.pod.namespace: kube-system
        k8-app: kube-dns
    toPorts:
     - ports:
       - port: "53"
         protocol: TCP
       - port: "53"
         protocol: UDP
       rules:
        dns:
          - matchPattern:"*"
```

1

```
  - toFQDNs:
    - matchPattern: "*.cilium.io"
```

2

An example Cilium Network Policy with two individual rules highlighted: (1) An egress rule that allows DNS records to be queried from the kube-dns component, and (2) A rule that allows non-DNS connections to be initiated to subdomains of cilium.io.

Considering egress DNS-based policies, in order for Cilium to allow traffic based on FQDNs, you must create two separate rules: Firstly, allow the DNS traffic that the application uses to resolve the domain names. This traffic must not only be allowed, but the network policy rule must also instruct Cilium to **learn DNS information**[36] through a "`rules: dns: ...`" statement, as demonstrated in the example. In addition to the DNS rule, you must also create another rule to match the traffic to the intended destination through use of a `toFQDNs` **egress**[37] or `fromFQDNs` **ingress**[38] statement.

Security considerations

When writing policies for FQDNs, it is critical that Cilium receives accurate information about the DNS name to IP mappings. Failure to provide accurate DNS information can result in the network policy allowing traffic to domain names that are backed by untrusted IP addresses. The common pattern to protect against this case in Kubernetes environments is to ensure that the DNS rule for interception only allows DNS requests from workloads towards the cluster's central DNS server. The cluster DNS server itself should only be trusted to hand out DNS mappings that it can attest.

Even if you are confident about the accuracy of DNS information that is provided to Cilium, note that Cilium FQDN policy only ensures that the traffic can reach an IP address that backs a known DNS name. FQDN rules do not validate host access at the target address at Layer 7. Consider if you may have infrastructure hosted on AWS S3, which allows connections to any S3 bucket name to reach a pool of servers that then further route the requests to an underlying bucket. When a request destined to an IP address backing `good-name.s3.amazonaws.com` hits the Cilium network policy layer, Cilium will enforce that the traffic is accessing an IP address legitimately backing that S3 bucket name. However, in the HTTP headers the request may specify the host as `bad-name.s3.amazonaws.com`. In this scenario, Cilium's `toFQDNs` rules will not restrict the traffic. In order to mitigate this scenario, you must implement SNI filtering[39] to restrict which server names are allowed by the network policy.

Debugging

Operating and debugging Cilium in general is a broad topic, so this section focuses specifically on debugging and verifying the operation of network policies. There will be two primary categories described below. Firstly, how to gain an overview of policy enforcement status by using centralized clusterwide tooling typically used in production environments.

[39] SNI filtering, also known as SNI whitelisting, only allows HTTP requests for specific server names. See the **feature status matrix**[40] for more details on feature support.

Secondly, this section will demonstrate per-node tooling which is more suitable for learning in a pre-production environment or evaluating against a **sysdump**[41] (system dump) snapshot which you have gathered from a pre-production or production environment.

Clusterwide tooling

The common first step for debugging any environment is to understand whether the cluster is in a good state, whether there are any active ongoing errors or warnings in the environment. The **Cilium CLI tool**[42] provides this information via `cilium status`:

```
$ cilium status
    /¯¯\
 /¯¯\__/¯¯\    Cilium:             OK
 \__/¯¯\__/    Operator:           OK
 /¯¯\__/¯¯\    Envoy DaemonSet:    disabled (using embedded mode)
 \__/¯¯\__/    Hubble Relay:       disabled
    \__/       ClusterMesh:        disabled
DaemonSet              cilium             Desired: 2, Ready: 2/2, Available: 2/2
Deployment             cilium-operator    Desired: 2, Ready: 2/2, Available: 2/2
Containers:            cilium             Running: 2
                       cilium-operator    Running: 2
Cluster Pods:          7/7 managed by Cilium
Helm chart version:
Image versions         cilium             quay.io/isovalent/cilium:v1.15.9-cee.1: 2
                       cilium-operator    quay.io/isovalent/operator-generic:v1.15.9-cee.1: 2
```

If there are active errors or warnings, then this tool displays them. This tool also provides the `sysdump` command which is useful when submitting a support request. The sysdump gathers a range of debugging information from the environment which can be used to investigate an incident. Note that if you are experiencing an issue and submit a support request, it is also important to highlight which workloads are experiencing the issue, and ideally provide reference IP addresses and/or nodes for those workloads. In large environments, it is also important to filter the sysdump to target just a few nodes - ideally at least one node actively experiencing an issue and one node not actively experiencing the issue.

Validating Network Policies

Network policies are stored in various different resource types, so you can check whether the relevant policies are installed correctly using `kubectl` as per the following example. Newer Cilium versions will also provide initial validation of network policies here which capture the majority of errors with the form and format of the policies[43].

[43] Cilium v1.16 introduced network policy validation for CNP and CCNP resources.

```
$ kubectl get --all-namespaces netpol,cnp,ccnp
NAMESPACE       NAME                                          AGE       VALID
default         ciliumnetworkpolicy.cilium.io/rule1           6h6m      True
```

You can further expand and inspect the policies through the `-o yaml` flag, which provides all of the detail about the policy including its metadata, specification, and status fields. KNP and CNP resources are namespaced, so they require the `-n` flag to specify the namespace to inspect. CCNP resources are not namespaced and do not require the use of this flag.

Cilium may also report issues in processing network policies into the cilium-agent logs. You can inspect the Cilium agent logs using "`kubectl logs -n kube-system ds/cilium`". Any network policy processing errors are printed with the level field set to warning or error. If a network policy cannot be successfully processed by the cilium-agent, then that agent will **reject the entire policy**, not just the rule which cannot be processed. This process of validating the network policies within the cilium-agent is more comprehensive and ultimately informs the enforcement posture, so it is recommended to consult the cilium-agent logs to confirm there are no errors or warnings indicating issues with enforcing network policies.

Other Resources

Cilium relies on other resources to implement policies, specifically the CiliumIdentity, CiliumEndpoint and CiliumEndpointSlice[44] objects. CiliumEndpoint resources provide a clear indication as to the association that Cilium has built between the individual workloads by namespace/name, the corresponding security identity, and the corresponding addresses. Given the security identity, you can find out which labels that Cilium takes into account when calculating network policy for those endpoints, either as the subject of a policy or as a peer.

```
$ kubectl get ciliumendpoints
NAME                                SECURITY IDENTITY     ENDPOINT STATE      IPV4 ADDRESS
deathstar-b4b8ccfb5-cszjn           13011                 ready               10.0.1.16
deathstar-b4b8ccfb5-fznr9           13011                 ready               10.0.0.233
tiefighter                          41254                 ready               10.0.0.147
xwing                               29954                 ready               10.0.0.150
$ kubectl get ciliumidentities 13011 -o yaml
apiVersion: cilium.io/v2
kind: CiliumIdentity
metadata:
 creationTimestamp: "2024-10-01T17:45:25Z"
```

[44] CiliumEndpointSlice is recommended for use primarily in environments with >500 nodes (See the **Isovalent scale guide**[45]).

```
  generation: 1
  labels:
    app.kubernetes.io/name: deathstar
    class: deathstar
    io.cilium.k8s.policy.cluster: default
    io.cilium.k8s.policy.serviceaccount: default
    io.kubernetes.pod.namespace: default
    org: empire
  name: "13011"
  resourceVersion: "1911"
  uid: e1514a11-4d88-4472-b8d6-b901c3a7d376
security-labels:
  k8s:app.kubernetes.io/name: deathstar
  k8s:class: deathstar
  k8s:io.cilium.k8s.namespace.labels.kubernetes.io/metadata.name: default
  k8s:io.cilium.k8s.policy.cluster: default
  k8s:io.cilium.k8s.policy.serviceaccount: default
  k8s:io.kubernetes.pod.namespace: default
  k8s:org: empire
```

This information is encoded in the central CRD store (or key-value store if configured), and is considered the “desired state” of the endpoints and identities. All nodes within the cluster pull these resources from the central store and use the information to decide how to apply network policy enforcement for connections to and from these managed endpoints.

Metrics

Cilium supports exporting various metrics through a Prometheus endpoint, which can be queried or assembled into dashboards, including the Isovalent Enterprise for Cilium Dashboards. Some notable metrics to consider from a network policy perspective include:

Metric name	Summary
cilium_bpf_map_pressure	Map pressure metric is labeled by `map_name` which may expose policy maps. By default the pressure is not exposed for policy maps, but if the pressure exceeds a threshold then Cilium will report these metrics. If the pressure approaches 100% then this should be investigated immediately to avoid production impact.

Metric name	Summary
cilium_drop_count_total	Packets dropped, labeled by various properties including `reason`. If the reason indicates a network policy enforcement reason, then traffic has been dropped as a result of the network policies. If the node is exposed to untrusted network endpoints, then this metric may increase based on unsolicited inbound connections. However, if the node should generally be connected only to trusted peers then sustained increases in this metric may be an indicator of connectivity issues, perhaps due to strict network policies. Investigate if experiencing connection disruption.
cilium_identity	Number of security identities known by this node, labeled by `type` which indicates identities that are shared between clusters, within the cluster, or local-only. If these numbers are in the thousands, consider reviewing the section **Limit Label Cardinality** as this can impact network availability (subject to how permissive your policies are).
cilium_policy_change_total	Number of policies processed, labeled by `outcome` (success or failure). Any failures should be investigated to determine enforcement status of network policies created in the environment. Failed policy changes may indicate Cilium is not protecting workloads due to a network policy misconfiguration.
cilium_policy_endpoint_enforcement_status	Enforcement status of Endpoints on the node, labeled by direction of enforcement. Depending on the configuration, the label `enforcement=none` includes the cilium-health simulated endpoint and the host endpoint, so up to a baseline value of 2. Determine your baseline enforcement status count in an environment with the same configuration and no workloads scheduled to the node. If the value of this metric is greater than your baseline value, then some workloads may not be actively protected by network policies. Endpoints protected on both ingress and egress by the network policy are counted with `enforcement=both`.

Metric name	Summary
cilium_policy_implementation_delay	Histogram of the propagation delay in seconds between Cilium receiving a network policy until the dataplane begins enforcing the policy. Use this metric to set expectations around how quickly new policies become effective, and to gauge the impact that scale has on the network policy enforcement engine. Consider optimization techniques for expressing network policy if the value is high.
cilium_policy_l7_total	Number of requests handled by Layer 7 introspection, labeled by the `proxy_type` and `rule`. The rule field highlights whether messages were forwarded, received, denied, or encountered parse errors. Monitor for high frequency of requests particularly if experiencing connectivity issues for paths selected by Layer 7 network policy rules.

Monitoring live traffic with Hubble

Isovalent Enterprise for Cilium provides a range of tooling around Hubble which provide visibility into live and optionally **historic flow data**[46] in your environment. Hubble can be useful to gather further information about the way that Cilium is handling network traffic via hubble metrics, a clusterwide CLI, or a web-based UI. For network operators, this tooling can provide similar functionality to traditional tools such as tcpdump while enhancing the flow data with Cilium-specific context such as the security identity of traffic and reasons for why traffic is dropped. This section introduces some of the monitoring capabilities, but for more information see the **Isovalent resources library**[47] for observability with Hubble.

```
$ hubble observe -f
Oct 1 10:38:30.087: default/client:57984 (ID:29954) <> default/server-b4b8ccfb5-cszjn:80
                    (ID:13011) Policy denied DROPPED (TCP Flags: SYN)
...
```

In this example, the traffic from the client towards the server is denied by policy, and the traffic is dropped. There are two kinds of “policy deny” verdicts: Explicit and implicit. Explicit deny verdicts are applied when there is a network policy with an `egressDeny` or `ingressDeny` statement which matches the traffic, causing Cilium to drop the traffic. These deny events present with the reason “Policy denied by denylist”. On the other hand, implicit deny events are triggered by a **Default Deny Posture** which drops all traffic that is not explicitly allowed. Implicit denies present with the message “Policy denied”, as seen in the example above.

The simplified view of "`hubble observe`" output provides timestamps, source and destination workloads, how the traffic was handled such as whether it was forwarded or dropped, and in the case of a drop, the reason why that traffic was dropped. The underlying flow data protocol however provides far more detailed information in json format, including IP addresses, node name, policies that allowed or denied the traffic[48], and more:


```
{
  "flow": {
  "time": "2024-10-02T02:57:40.311366786Z",
  "uuid": "85b4621d-7760-40e0-8859-21ab6b9b16a3",
  "verdict": "FORWARDED",
  "ethernet": {
    "source": "f6:5f:5d:76:20:0e",
    "destination": "3a:36:80:b6:d5:63"
   },
     "IP": {
       "source": "10.0.0.147",
       "destination": "10.0.1.16",
       "ipVersion": "IPv4"
   },
    "l4": {
     "TCP": {
      "source_port": 49764,
      "destination_port": 80,
      "flags": {
        "SYN": true
       }
     }
   },
   "source": {
     "identity": 41254,
     "cluster_name": "default",
     "namespace": "default",
     "labels": [
        "k8s:app.kubernetes.io/name=tiefighter",
        "k8s:class=tiefighter",
        "k8s:io.cilium.k8s.namespace.labels.kubernetes.io/metadata.name=default",
        "k8s:io.cilium.k8s.policy.cluster=default",
        "k8s:io.cilium.k8s.policy.serviceaccount=default",
        "k8s:io.kubernetes.pod.namespace=default",
        "k8s:org=empire"
   ],
     "pod_name": "tiefighter"
   },
"destination": {
   "ID": 112,
   "identity": 13011,
   "namespace": "default",
```


[48] Network Policy correlation for "allow" rules is available in all supported versions. This was extended to support explicit "deny" rules in Cilium v1.16.

```
    "labels": [
       "k8s:app.kubernetes.io/name=deathstar",
       "k8s:class=deathstar",
       "k8s:io.cilium.k8s.namespace.labels.kubernetes.io/metadata.name=default",
       "k8s:io.cilium.k8s.policy.cluster=default",
       "k8s:io.cilium.k8s.policy.serviceaccount=default",
       "k8s:io.kubernetes.pod.namespace=default",
       "k8s:org=empire"
    ],
    "pod_name": "deathstar-b4b8ccfb5-cszjn",
    "workloads": [
     {
       "name": "deathstar",
       "kind": "Deployment"
      }
    ]
},
"Type": "L3_L4",
    "node_name": "kind-control-plane",
    "node_labels": [
       "beta.kubernetes.io/arch=amd64",
       "beta.kubernetes.io/os=linux",
       "kubernetes.io/arch=amd64",
       "kubernetes.io/hostname=kind-control-plane",
       "kubernetes.io/os=linux",
       "node-role.kubernetes.io/control-plane=",
       "node.kubernetes.io/exclude-from-external-load-balancers="
],
    "event_type": {
      "type": 5
},
"traffic_direction": "INGRESS",
"policy_match_type": 2,
"is_reply": false,
"Summary": "TCP Flags: SYN",
"ingress_allowed_by": [
    {
      "name": "rule1",
      "namespace": "default",
      "labels": [
         "k8s:io.cilium.k8s.policy.derived-from=CiliumNetworkPolicy",
         "k8s:io.cilium.k8s.policy.name=rule1",
         "k8s:io.cilium.k8s.policy.namespace=default",
         "k8s:io.cilium.k8s.policy.uid=0ddfc37f-bad9-4b47-a9f1-361126e2af2f"
    ],
    "revision": "2"
    }
  ]
},
"node_name": "kind-control-plane",
"time": "2024-10-01T10:57:40.311366786Z"
}
```

Per-Node Tooling

There is a range of tooling shipped with the cilium-agent container running on each node which allows deeper introspection of the local state. This information can be gathered using a sysdump from the cluster-wide CLI, or by opening a shell in a Cilium Pod. The `cilium-dbg` tool in the Cilium container image can query similar information as the cluster-wide tooling, but provides more detailed representations of the state at the node level. This subsection provides some useful command output examples for understanding how Cilium implements network policies. Consider comparing the output of these commands from a sysdump in different nodes if you are investigating a policy issue that presents differently on different nodes.

Local Status and Endpoints

The status output at the node level provides details about software modules operating within the Cilium agent. This includes the status of the host firewall on the node and the status of the proxy:

```
root@kind-control-plane:/home/cilium# cilium-dbg status
KVStore:                 Ok   Disabled
Kubernetes:              Ok   1.29 (v1.29.2) [linux/amd64]
Kubernetes APIs:         ["EndpointSliceOrEndpoint", "cilium/v2::CiliumClusterwideNetworkPolicy",
"cilium/v2::CiliumEndpoint",        "cilium/v2::CiliumNetworkPolicy",       "cilium/v2::CiliumNode",
"cilium/v2alpha1::CiliumCIDRGroup",     "core/v1::Namespace",   "core/v1::Pods",   "core/v1::Service",
"networking.k8s.io/v1::NetworkPolicy"]
KubeProxyReplacement:    False    [eth0      172.18.0.3 fc00:c111::3 fe80::42:acff:fe12:3]
Host firewall:           Disabled
SRv6:                    Disabled
CNI Chaining:            none
CNI Config file:                            successfully wrote CNI configuration file to
/host/etc/cni/net.d/05-cilium.conflist
Cilium: Ok 1.15.9-cee.1 (v1.15.9-cee.1-a2a2b7ad)
NodeMonitor:             Listening for events on 12 CPUs with 64x4096 of shared memory
Cilium health daemon:    Ok
IPAM:                    IPv4: 3/254 allocated from 10.0.1.0/24,
IPv4 BIG TCP:            Disabled
IPv6 BIG TCP:            Disabled
BandwidthManager:        Disabled
Host Routing:            Legacy
Masquerading:            IPTables [IPv4: Enabled, IPv6: Disabled]
Controller Status:       28/28 healthy
Proxy Status:            OK, ip 10.0.1.77, 0 redirects active on ports 10000-20000, Envoy: embedded
Global Identity Range:   min 256, max 65535
Hubble:                  OK, Current/Max Flows: 1693/4095 (41.34%), Flows/s: 0.48 Metrics: Disabled
Encryption:              Disabled
Cluster health:          2/2 reachable (2024-10-01T10:34:34Z)
Modules Health:          Stopped(0) Degraded(0) OK(11)
```

The list of endpoints furthermore lists the policy enforcement status for both ingress and egress for each endpoint on the node. Note that an Endpoint on a node has an *Endpoint ID*, which is only unique within the current node and is not shared with other nodes in the cluster. The Security Identity on the other hand is shared across nodes and the number is reused by all workloads with the same identity-relevant labels.

```
root@kind-control-plane:/home/cilium# cilium-dbg endpoint list
ENDPOINT POLICY (ingress) POLICY (egress) IDENTITY LABELS (source:key[=value])  IPv4         STATUS
         ENFORCEMENT      ENFORCEMENT
112      Enabled          Disabled         13011    ...                          10.0.1.16    ready
1336     Disabled         Disabled         4        reserved:health              10.0.1.170   ready
2132     Disabled         Disabled         1        reserved:host                             ready
```

Inspecting Local Policy State

Given a locally deployed endpoint from the prior commands, you can inspect the underlying eBPF policy that is currently applied to see exactly which traffic will be allowed by the policy. This lists the type of policy (allow or deny), the peer (in this case listed with the labels, though there is a `--numeric` option to list the numeric security identity), port/protocol, as well as other information like how many packets and bytes have hit the rule. In this case, there is a set of network policies denying and allowing specific traffic on ingress with a default deny, and there is an implicit allow for all egress. The egress allow is denoted with a labels match for `reserved:unknown`, which represents an allow for "any" traffic.[49]

```
root@kind-control-plane:/home/cilium# cilium-dbg bpf policy get 112
POLICY DIRECTION LABELS (source:key[=value])           PORT/PROTO       PROXY PORT   AUTH TYPE  BYTES  PKTS
Deny   Ingress   k8s:app.kubernetes.io/name=xwing      80/TCP           NONE         disabled   1628     22
                  ...
Allow   Ingress  reserved:host                        ANY              NONE         disabled   0        0
                  ...
Allow   Ingress  k8s:app.kubernetes.io/name=deathstar  80/TCP           NONE         disabled   0        0
                  ...
Allow   Ingress  k8s:app.kubernetes.io/name=tiefighter 80/TCP           NONE         disabled   2088     2
                ...
Allow   Egress   reserved:unknown                      ANY              NONE         disabled   0        0
```

Information about security identities are largely the same as the clusterwide view, but this command may list additional security identities that are not listed in the equivalent `kubectl` command. Some security identities are only allocated locally within the node, and this view will list all such security identities that are not synchronized across the cluster. These security identities typically start with "16777".

[49] This is planned to change to display ANY for consistency with the PORT/PROTO field (See GitHub Issue #35556[50]).

```
root@kind-worker:/home/cilium# cilium-dbg identity list
ID              LABELS
1               reserved:host
2               reserved:world
3               reserved:unmanaged
4               reserved:health
5               reserved:init
6               reserved:remote-node
7               reserved:kube-apiserver
                reserved:remote-node
8               reserved:ingress
9               reserved:world-ipv4
10              reserved:world-ipv6
7620            k8s:class=mediabot
                k8s:io.cilium.k8s.namespace.labels.kubernetes.io/metadata.name=default
                k8s:io.cilium.k8s.policy.cluster=default
                k8s:io.cilium.k8s.policy.serviceaccount=default
                k8s:io.kubernetes.pod.namespace=default
                k8s:org=empire
...
16777217        cidr:192.0.2.3/32
                reserved:world
```

Scaling

Isovalent performs scalability testing of Isovalent Enterprise for Cilium to validate the behavior of the components when pushing the limits of different resources to high scale, such as:

- Thousands of Nodes
- Tens of thousands of Endpoints
- Thousands of network policies
- Thousands of Security Identities
- Dozens of workloads being created per second across the cluster

Provisioning Cilium for high scale requires careful consideration of a range of configuration parameters which are described more generally in the **Isovalent Scalability Guide**[51]. This section is specifically focused on considering the dimensions that affect Cilium's ability to distribute, calculate and enforce network policies on individual nodes. This will be followed by recommendations for configuring Cilium to optimally implement network policies in high-scale environments.

Scalability Factors

The following table highlights the primary factors influencing the scalability of network policy enforcement in a Cilium environment for the central control plane (affecting cluster-wide state distribution), the local control plane (affecting policy implementation delay), and the data plane (enforcement on each connection).

Central Control Plane	Local Control Plane	Data Plane (eBPF)

• Nodes
• Security Identities
• Endpoints in cluster
• Endpoint churn
• Network Policies
	• Security Identities
	• Network Policies
	• Locally scheduled Endpoints
	• Policies with local subjects
	  ○ Total number of rules
	  ○ Use of broad selectors
	  ○ Use of explicit Deny rules
	• Unique selectors in policies
		• Security Identities
		• Endpoints in cluster
		• CIDR rules
		• Concurrent connections
		• Use of broad selectors
		  ○ Wildcard Selectors
		  ○ Cluster entity
		• Use of Layer 7 rules

Central Control Plane

The central control plane is composed of the Kubernetes control plane and the Cilium Operator. Its scalability is primarily influenced by the total number of resources in the environment and the total number of nodes that must receive that information. This information is distributed through a central data store, which can be backed by Kubernetes CRDs or a Key-Value store (Etcd). The central control plane is responsible for processing and distributing these events to each node in the cluster. As the total number of resources distributed to each node in the cluster increases, the CPU, memory, and network resource usage of the central control plane will increase as well. In particular, the rate of updates for the Security Identity and Network Policy resources have a high impact on resource utilization. Furthermore, resource usage can be significantly impacted by frequent churn on the number of workloads being created and destroyed.

Local Control Plane

The scalability of the local control plane is also affected by the total number of Security Identities and Network Policies, but to a lesser extent than the central control plane. This is because it doesn't need to take into account how other nodes are processing the data. Instead, the local control plane depends more heavily on the number of workloads scheduled to the local node and the policies applying to them. Each node processes updates for Security Identities and Network Policies, but if these updates do not affect the local workloads then the impact to the local node is primarily limited to marshaling costs. For instance, if a network policy has a subject selector that selects labels that are not applied to a workload on the node, then Cilium will parse and store the rules internally but it will not perform subsequent calculations at that time.

For network policies that do apply to workloads present on the node, Cilium is optimized to calculate the policy once for each Security Identity which is used by a local workload and share this policy calculation result with other instances of the same workload on the node. The calculation itself depends on the number of individual rules in the network policies applying to local workloads, and is impacted by the content of the rules such as the use of Layer 7 features.

When the Cilium Agent is restarted or upgraded on each node, upon startup it fetches all resources that it needs to implement policy from the central control plane. We recommend **limiting the number of concurrent unavailable agents**[52] in the cluster to prevent this from causing the central control plane to become overloaded. In practice, we expect that even in large environments with thousands of Security Identities and tens of thousands of network policies to synchronize, Cilium Agent restart should complete within a few minutes.

Data Plane (eBPF)

Similar to the central control plane, the data plane on each node is primarily impacted by the total number of resources used for policy calculation and enforcement. This is driven primarily by configurable limits on the capacity (and hence memory usage) of different components of the data plane, such as the IP to Identity maps, the connection tracking maps, and the policy maps. The most common outcome from exceeding the capacity limits on these maps is that traffic is dropped when the policy specifies that the traffic should be allowed. These outcomes and mitigations will be explored in the following section. Most of the scalability impacts on the data plane layer are based on the capacity limits (memory), as the Cilium design optimizes for minimal network packet latency. If memory usage is a primary concern, Cilium can be tuned to use less memory, potentially at the expense of network performance.

If there are Layer 7 rules in the network policies, then these rules typically require deeper processing in a userspace proxy. Forwarding this traffic into the proxy, and the more computationally expensive features implemented by the proxy, can cause additional latency on traffic affected by such policies. When writing Layer 7 rules statements, it is advisable to minimize the scope of these rules to ensure that only the traffic destined to a specific endpoint is subject to this additional processing.

Guidance

For each Security Identity that is selected by a rule in a policy, typically around one eBPF rule will be generated into a policy map. This can vary depending on the use of overlapping selectors, in particular CIDR selectors and application port selectors, extending to several dozen rules in one of the more severe cases[53]. However, policy rules are not defined by which Security Identity the policy should govern, they are instead defined by the selectors which the labels of the security identities that can be allowed. As such, a single rule in a network policy can generate many eBPF rules - in the worst case, selecting each Security Identity in the entire cluster.

Therefore, when considering how to optimize policies for scale, you should take into account not just the number of policies, but also how permissive those policies are, as well as the total number of Security Identities. The following sections will provide some general guidelines about building optimal policies, which should be used to supplement your own testing in staging environments prior to deployment into production.

Simpler is Better

Generally speaking, the less variation between the rules, the more efficient Cilium can be while evaluating the network policies:

- If a rule can be implemented in a single `CiliumClusterwideNetworkPolicy`, then that is more efficient than defining multiple copies of the rule in `NetworkPolicy` or `CiliumNetworkPolicy` resources in different namespaces.
- For each peer selector, it is more efficient if the selector applies to only a few security identities rather than many security identities (so it is better to avoid the “cluster” entity selector or wildcard selector “`matchLabels: {}`” where possible).
- If the same selector is reused in multiple rules and it selects few peers, then it may be more efficient to reuse this selector rather than to declare unique selectors in each rule.

If there are already many rules and security identities, then introducing Deny policies may further extend the policy calculation time and the total number of eBPF map entries to express the policy. For each Deny rule, when Cilium calculates the eBPF rule state, Cilium identifies any conflicting Allow rules and removes them in order to create a “flat”, largely non-overlapping set of policy rules in the eBPF maps. Therefore, if it is possible, express your network policy using a minimal set of explicit Allow rules as a permissive policy with a default deny, as this is preferred over a set of explicit Deny rules with a default allow.

[53] Broad CIDR Selectors with narrow except clauses can generate a range of rules up to the bit-length of the IP family used in the CIDR rule. Furthermore, rules with port ranges can translate into several eBPF policy map entries, but these are sublinear with respect to the total number of ports specified in the rule.

Use Permissive Policies with Caution

Cilium Policy Map limits are by default optimized for highly restrictive network policies that only allow a relatively small number of Security Identities to communicate with the subject. This reflects both the most efficient representation of the policy, as well as the most secure security posture for the endpoint. However, in some cases it can be useful to expose a workload to allow connections with a large number of remote peers. Some such cases are also optimized, such as when including a selector for the "all" entity which allows traffic to any destination. However, there are some notable selectors which can be inefficient to process and encode in eBPF maps. For example, the following peer selectors can result in high cardinality based on the number of Security Identities in the cluster[54]:

- An endpoint selector that selects many identities, such as `matchLabels: {}`, particularly when used in CiliumClusterwideNetworkPolicy.
- The cluster entity selector, which selects every Cilium-managed workload and node.

When operating in a high scale environment, these selectors scale based on the total number of Security Identities defined in the cluster. If this number exceeds the size of the policy map, then Cilium may not be able to encode the policies into the map, resulting in the network policy enforcement diverging from the desired policy. It is therefore important to monitor policy map limits to be able to identify and mitigate this situation before it affects applications.

Monitor Policy Map Limits

If you have written permissive network policy rules that select a very large number of Security Identities and this fills up the entire eBPF policy map, then you may begin to experience application impact for workloads that are subject to the permissive policies. When using Allow policies, this typically results in one of the "allow rules" not being created in the data plane, so any traffic that should otherwise be allowed by the rule will instead be dropped due to the lack of a matching rule. This can result in connectivity disruption to workloads that communicate using the affected policy. If Cilium cannot insert a Deny policy into a full policy map, traffic may be allowed instead[55]. Therefore it is highly recommended to have sufficient monitoring and alerts in place to track the `bpf_map_pressure` metrics highlighted under **Metrics**. This monitoring will signal that you should review the selectors used in network policies and to avoid selectors that are broadly permissive. If you experience this problem, you can mitigate the immediate effects by increasing the size of the policy map[56].

[54] These cases are known to introduce high cardinality as of Cilium v1.16. Isovalent periodically evaluates the efficiency of policy statements in order to optimize them for common use cases.

[55] This issue only affects explicit deny rules when the policy map capacity is exceeded. As of 2024, an advanced "lockdown" mode is on the roadmap which would isolate affected workloads in this condition.

[56] The **eBPF Maps**[57] documentation describes how to to configure eBPF map sizes.

Limit Label Cardinality

The total number of Security Identities is a major factor in the scalability of network policy in a Cilium environment. Within an individual cluster, the maximum number of Security Identities for workloads in the cluster is limited to around 65,000 by default. Each additional cluster has its own allocation space of another 65,000 cluster-specific Security Identities. In addition to this limit, each node manages its own additional pool of around 16 million Security Identities that are used solely within the node for local enforcement of network policies. This additional pool is not synchronized with other nodes in the cluster. Considering the size of these Security Identity allocation pools, if there are many security identities allocated and they are selected by permissive policies, that may lead to considerable pressure on the eBPF policy maps. If you can reduce the total number of Security Identities, then this relieves pressure on the maps.

The optimal configuration of labels is for the minimal set of **Identity-relevant Labels** that can still express the desired security relationships between applications in the environment. For instance, if you write network policy based solely on the source and destination Kubernetes namespace, then the Pod labels could be ignored for security purposes. In that case you can configure the Identity-Relevant Labels to rely purely on the namespaces to determine the security identity of the applications. This setting allows you to trade off the granularity of network policy enforcement for performance at scale, so it is recommended to consider which labels are critical for your network enforcement posture and configure identity-relevant labels accordingly.

Account for Highly Dynamic Services

Environments with a high degree of scale are often the most dynamic environments as well. As you continue to monitor and operate such environments, it is important to keep a close eye on both the sheer number of different types of resources, but also how frequently they change. In addition to the workload and network policy churn described earlier, another challenge that many Cilium users experience is how to handle the high churn involved in managing highly dynamic domain names, notably when combining `ToFQDNs` policies with Amazon S3. Applications that are designed to leverage DNS multi-value answers (MVA) put additional pressure on Cilium to track the IP-to-FQDN associations to enforce network policies. Considering a case with up to eight IP addresses per query, with DNS TTLs set to 5s, it's plausible for workloads querying a single domain to associate up to 96 unique IPs with the domain name per minute.

The effect that this has on Cilium policy enforcement may be exacerbated by high workload density, particularly with multiple applications with frequent DNS resolution and high connection rates. There is ongoing effort to identify and optimize for these scenarios through better metrics tooling, tunable options, and policy optimizations[58]. For the latest on these efforts, consult with your Solutions Architect.

[58] Cilium's Security Identity allocation mechanism in Cilium v1.16 was optimized to better account for domains with highly dynamic IP ranges such as Amazon S3.

Conclusion

In a world of highly dynamic workloads, the network infrastructure becomes increasingly important to protect the data communicated between applications. It becomes difficult to use traditional tooling to ensure the proper enforcement of network policy and ensure that enforcement follows those increasingly ephemeral workloads. Cilium was designed from the ground up as a cloud-native solution to these problems using modern solutions to solve classic networking problems. While this requires taking a fresh look at network policy enforcement, this provides a range of benefits to security and efficiency over existing solutions. This document has covered the base concepts used by Cilium for network policy, applied those concepts to the Cilium architecture, and explored how to leverage that knowledge to write, debug and operate network policies in your environment at scale.

This deep dive is intended to serve as a stepping stone towards deeper discussions with the team members that you will rely on to support Cilium in your production environments. For a topic as detailed and nuanced as network policy enforcement, it's a tall order to cover the topic for a range of readers with a good balance of breadth and depth. If you have questions coming out of this document, please don't hesitate to reach out to your local Cilium experts to discuss your questions and concerns. We hope that this document will help to facilitate those conversations by providing you with a shared understanding and language around the enforcement of network policies.

This document is one of many available in the **Isovalent Resources Library**[59] and **Isovalent Product Documentation**[60] for Cilium, Tetragon and eBPF. In order to further solidify this knowledge, we recommend exploring the **Isovalent Labs**[61], in particular *Isovalent Enterprise for Cilium: Network Policies*.

Appendix

[3] https://docs.cilium.io/en/stable/gettingstarted/terminology/#special-identities

[4] https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/

[5] https://docs.cilium.io/en/stable/security/host-firewall/#host-firewall

[7] https://docs.cilium.io/en/stable/security/policy/language/#node-based

[8] https://docs.isovalent.com/operations-guide/features/status.html

[9] https://docs.cilium.io/en/stable/security/policy/language/#entities-based

[11] https://docs.cilium.io/en/stable/security/policy/intro/#policy-enforcement-modes

[13] https://docs.cilium.io/en/stable/security/policy/language/#cidr-select-nodes

[15] https://docs.isovalent.com/operations-guide/features/fqdn-ingress-network-policy/introduction.html

[16] https://docs.cilium.io/en/stable/security/policy/language/#dns-based

[17] https://docs.cilium.io/en/stable/security/aws/#policy-language

[18] https://docs.cilium.io/en/stable/security/policy/language/#services-based

[20] https://docs.isovalent.com/operations-guide/features/status.html

[22] https://docs.isovalent.com/operations-guide/features/status.html

[23] https://docs.isovalent.com/operations-guide/features/fqdn-ingress-network-policy/introduction.html

[24] https://docs.isovalent.com/operations-guide/features/fqdn-ingress-network-policy/implementation-details.html#ciliumcidrgroup

[25] https://docs.isovalent.com/v1.15/operations-guide/monitoring/sysdump/index.html

[26] https://docs.cilium.io/en/stable/network/kubernetes/kubeproxy-free/#client-source-ip-preservation

[27] https://docs.cilium.io/en/stable/network/concepts/routing/#encapsulation

[28] https://isovalent.com/blog/post/2022-05-03-servicemesh-security/

[29] https://isovalent.com/resource-library/labs/?solution=security

[33] https://docs.isovalent.com/operations-guide/features/status.html#network-policy

[34] https://docs.isovalent.com/operations-guide/features/status.html#network-policy

[35] https://kubernetes.io/docs/concepts/services-networking/network-policies/

[36] https://docs.cilium.io/en/stable/security/policy/language/#obtaining-dns-data-for-use-by-tofqdns

[37] https://docs.cilium.io/en/stable/security/policy/language/#dns-based

[38] https://docs.isovalent.com/operations-guide/features/fqdn-ingress-network-policy/introduction.html

[40] https://docs.isovalent.com/operations-guide/features/status.html#network-policy

[41] https://docs.isovalent.com/operations-guide/monitoring/sysdump/index.html

Appendix

[42] https://docs.isovalent.com/operations-guide/monitoring/sysdump/index.html

[45] https://docs.isovalent.com/operations-guide/scalability/plan-for-scalability.html

[46] https://docs.isovalent.com/operations-guide/hubble/index.html

[47] https://isovalent.com/resource-library/?solution=observability

[50] https://github.com/cilium/cilium/issues/35556

[51] https://docs.isovalent.com/operations-guide/scalability

[52] https://docs.isovalent.com/operations-guide/scalability/large-scale-cluster.html#upgrading-cilium

[57] https://docs.cilium.io/en/stable/network/ebpf/maps/#ebpf-maps

[59] https://isovalent.com/resource-library/

[60] http://docs.isovalent.com

[61] https://isovalent.com/resource-library/labs/?solution=security

ISOVALENT
now part of cisco

Cilium Network Policy Deep Dive

www.isovalent.com