

Politecnico di Torino

Master's Degree Course in Cybersecurity

Master's Degree Thesis

Kubernetes Misconfigurations Across RBAC and NetworkPolicy: A Taxonomy and its Practical Validation

---

Supervisors:
Fulvio Valenza
Daniele Bringhenti
Riccardo Sisto
Francesco Pizzato

Candidate:
Simone Di Nucci

Academic Year 2025–2026

Abstract

Kubernetes governs what a workload can do through Role-Based Access Control (RBAC) and what it can reach through NetworkPolicy. The two are configured separately, checked by separate tools and, as this thesis shows, studied separately in the literature too.

This work builds a taxonomy of misconfigurations for each mechanism on its own and then for the way the two interact. RBAC and NetworkPolicy come first, each derived bottom up from the detection rules that security tools already apply and grounded in CVEs verified by hand. The result is five RBAC families covering seventeen patterns and six NetworkPolicy families covering fifteen. Both are organised the same way, first by the property that fails and then by the mechanism behind the failure, which is what makes them directly comparable.

A third taxonomy covers what only appears when RBAC and NetworkPolicy are read together. Most of these are compositions. An RBAC weakness and a NetworkPolicy weakness that do nothing on their own become dangerous once combined. Two are different and they rely on one mechanism directly influencing the other one. For example, a single RBAC permission changes the network configuration directly and disables an isolation that was otherwise written correctly. Neither shape is visible to a tool that only looks at one plane.

Every cross-domain pattern found this way, except one held back as a borderline foundation condition rather than an attacker capability, was tested in practice by reproducing it in a running cluster, then run again against the fix that should stop it. Eight patterns remain valid under this test, one does not and was dropped from the taxonomy, and the borderline condition was retained separately without being reproduced in the lab. The same test, run with the outcome reversed, confirms the pairings that were ruled out from the start. This yields a matrix of thirty family pairings where every cell is either a validated cross-pattern or a stated reason why it isn't one. The plain conclusion is that the tools surveyed look at one plane at a time, so none of them can see an attack path that lives in the combination of the two.

Acknowledgements

Contents

Abstract i

Acknowledgements ii

1 Introduction 1
1.1 Thesis structure . . . 1

2 Background 3
2.1 Cluster architecture . . . 3
2.2 Core objects . . . 4
2.3 The two security mechanisms . . . 6
2.4 Why these two mechanisms . . . 7

3 Motivation 8
3.1 Problem statement . . . 8
3.2 Thesis objective . . . 8

4 Methodology and Dataset 10
4.1 Organising principles . . . 10
4.2 The collection method . . . 14
4.3 Sources . . . 20
4.3.1 RBAC sources . . . 20
4.3.2 NetworkPolicy sources . . . 20
4.4 The two datasets . . . 21
4.5 Scope and domain distinctions . . . 21

5 RBAC Misconfigurations 23
5.1 Corpus statistics . . . 24
5.2 Family R-F1 — Identity Overexposure . . . 24
5.2.1 R-P1.1 — Cluster-admin binding . . . 25
5.2.2 R-P1.2 — System groups binding . . . 28
5.2.3 R-P1.3 — Privileged SA distribution . . . 31
5.3 Family R-F2 — Permission Overreach . . . 34
5.3.1 R-P2.1 — Wildcard permissions . . . 36
5.3.2 R-P2.2 — Manage-all over a scope . . . 38
5.3.3 R-P2.3 — Aggregated ClusterRole abuse . . . 40
5.4 Family R-F3 — Sensitive Resource Exposure . . . 42
5.4.1 R-P3.1 — Secret access . . . 43
5.4.2 R-P3.2 — Config and observability access . . . 46

5.5 Family R-F4 — Privilege Escalation Paths . . . 49
5.5.1 R-P4.1 — RBAC escalation verbs . . . 51
5.5.2 R-P4.2 — Credential issuance . . . 54
5.5.3 R-P4.3 — Admission-webhook control . . . 57
5.5.4 R-P4.4 — Workload creation . . . 59
5.5.5 R-P4.5 — Pod execution control . . . 61
5.5.6 R-P4.6 — Cluster substrate access . . . 63
5.6 Family R-F5 — Authorization Layer Preconditions . . . 68
5.6.1 R-P5.1 — RBAC authorizer disabled or bypassed . . . 69
5.6.2 R-P5.2 — Node authorizer missing . . . 73
5.6.3 R-P5.3 — Controller-manager SA misconfig . . . 74

6 NetworkPolicy Misconfigurations 75
6.1 Corpus statistics . . . 75
6.2 Family N-F1 — Missing Isolation . . . 76
6.2.1 N-P1.1 — Missing default-deny baseline . . . 78
6.2.2 N-P1.2 — Missing baseline on new namespaces . . . 81
6.3 Family N-F2 — Over-permissive Connectivity . . . 82
6.3.1 N-P2.1 — Over-broad selector . . . 83
6.3.2 N-P2.2 — Unrestricted ipBlock `0.0.0.0/0` . . . 87
6.3.3 N-P2.3 — Wrong selector composition . . . 89
6.3.4 N-P2.4 — Over-permissive port exposure . . . 90
6.4 Family N-F3 — Ineffective Policy . . . 93
6.4.1 N-P3.1 — Dangling selector . . . 94
6.4.2 N-P3.2 — Missing `policyTypes` direction . . . 96
6.4.3 N-P3.3 — Misread deny semantics . . . 97
6.5 Family N-F4 — Sensitive Endpoint Exposure . . . 97
6.5.1 N-P4.1 — Sensitive endpoint reachable . . . 99
6.6 Family N-F5 — Enforcement Layer Preconditions . . . 101
6.6.1 N-P5.1 — NetworkPolicy not enforced by CNI . . . 102
6.6.2 N-P5.2 — Host-network bypass . . . 103

7 Cross-Domain Misconfigurations 106
7.1 What a cross-misconfiguration is . . . 106
7.2 Why this taxonomy cannot be built bottom up . . . 107
7.3 Construction method: compose, formalise, validate . . . 108
7.4 The RBAC × NetworkPolicy composition matrix . . . 109
7.5 The five cross-families . . . 117
7.5.1 **XF1** — Lateral Movement . . . 117
7.5.2 **XF2** — Data Exfiltration . . . 119
7.5.3 **XF3** — Cross-Boundary Access . . . 121
7.5.4 **XF4** — Defense Evasion via the Network Control Surface . . . 123
7.5.5 **XF5** — Authorizer Disabled and Substrate Not Enforcing . . . 125
7.6 Scope, limits and summary . . . 126

8 Practical Validation **128**
8.1 Method . . . 128
8.2 Two cases in full . . . 130
8.3 Results . . . 134
8.4 Threats to validity . . . 138
8.5 Summary . . . 138

9 Conclusions **140**
9.1 What was built . . . 140
9.2 What the results say . . . 140
9.3 Limits . . . 141
9.4 Future work . . . 142

A Family N-F6 — CNI-extension & Admin Policy **144**
A.1 N-P6.1 — Clusterwide override too broad . . . 145
A.2 N-P6.2 — Precedence Inversion . . . 146
A.3 N-P6.3 — Admin baseline over-permissive . . . 148

B Manually verified CVEs **150**
B.1 RBAC domain . . . 150
B.2 NetworkPolicy domain . . . 152

C The validation lab environment **153**

List of Figures

2.1 Cluster architecture 4
2.2 Core objects and their relations 5
2.3 RBAC and NetworkPolicy protect different paths 6

4.1 The two single-domain taxonomies share one shape. 14
4.2 From a detection rule to a taxonomy pattern 15

7.1 Attack chain for the X1 composition 119

8.1 The three validation designs 129
8.2 Objects deployed for a composition test 130
8.3 Why exec bypasses NetworkPolicy 135

List of Tables

4.1 Security property each family guards . . . 12
4.2 How an entry is assigned to a family . . . 13
4.3 The entry schema . . . 16
4.4 One entry, end to end . . . 17
4.5 The classification scoring scheme (RBAC collector). . . . 18
4.6 Verified CVEs by family . . . 19
4.7 RBAC: nine collection sources. . . . 20
4.8 NetworkPolicy: additional sources on top of the shared RBAC dataset. 21
4.9 Summary of the two datasets . . . 21

5.1 Data funnel for the RBAC domain . . . 24

6.1 Data funnel for the NetworkPolicy domain . . . 76

7.1 Composition matrix: RBAC against NetworkPolicy families . . . 110
7.2 Row **R-F1** against every NetPol family . . . 111
7.3 Row **R-F2** against every NetPol family . . . 111
7.4 Row **R-F3** against every NetPol family . . . 112
7.5 Row **R-F4** against every NetPol family . . . 113
7.6 Row **R-F5** against every NetPol family . . . 114
7.7 Cells that collapse into an existing pattern . . . 116
7.8 Schema for presenting a cross-pattern . . . 117
7.9 The cross-taxonomy at a glance . . . 127

8.1 Practical validation of the cross-domain matrix . . . 138

9.1 Summary of the three taxonomies . . . 140

B.1 CVEs verified for the RBAC taxonomy . . . 151
B.2 CVEs verified for the NetworkPolicy taxonomy . . . 152

Chapter 1

Introduction

Kubernetes has become the standard platform for deploying and orchestrating containerised applications. Two mechanisms together define the security of a cluster: Role-Based Access Control (RBAC), which decides what an identity is allowed to do, and NetworkPolicy, which decides which pods a workload is allowed to reach. The two are configured, enforced and normally audited separately and the main focus of this thesis is about what that separation hides.

1.1 Thesis structure

The rest of the thesis is organised as follows.

- Chapter 2 gives the background needed to follow the rest of the thesis, the architecture of a Kubernetes cluster and how RBAC and NetworkPolicy work.
- Chapter 3 explains why analysing RBAC and NetworkPolicy in isolation misses part of the picture, states the problem this thesis addresses, and sets out the thesis objective.
- Chapter 4 describes the methodology shared by the RBAC and NetworkPolicy taxonomies, how the data were collected and how entries are classified.
- Chapter 5 presents the RBAC taxonomy.
- Chapter 6 presents the NetworkPolicy taxonomy.
- Chapter 7 presents the cross-domain taxonomy, the layer where RBAC and NetworkPolicy meet. It explains why this layer needs a different method from the other two and presents the cross-patterns.
- Chapter 8 tests the cross-patterns in a running cluster.
- Chapter 9 draws together what the three taxonomies show and what they leave open.

Three appendices close the thesis.

- Appendix A develops the one NetworkPolicy family that rests on analysis rather than on evidence from the dataset.

- Appendix B lists every CVE checked by hand, including those not quoted in a pattern box.
- Appendix C describes the lab environment used for the validation in Chapter 8.

Chapter 2

Background

Kubernetes is an open-source platform used to deploy, manage and orchestrate containerised applications across multiple machines. Rather than managing each container manually, administrators define the desired state of the system through configuration files, while Kubernetes continuously works to ensure that the actual state matches the desired one. For example, it can automatically schedule containers on available machines, restart them if they fail, or replace them during updates.

Thanks to this declarative approach, Kubernetes has become the standard solution for container orchestration and at the same time, it introduces new security challenges, since the security boundary is no longer limited to a single machine but extends to the entire cluster.

This chapter provides the background needed to understand the rest of this thesis. It first describes the architecture of a Kubernetes cluster and its main components and then introduces the two security mechanisms that are the focus of this work: Role-Based Access Control (RBAC) and NetworkPolicy.

2.1 Cluster architecture

A Kubernetes cluster consists of a group of machines, called nodes, that work together under the coordination of a control plane.

The control plane is responsible for managing the entire cluster and making decisions about its operation and its main component is the API server, which acts as the central access point for all communication with the cluster. Every Kubernetes API request, whether it comes from an administrator, an internal controller, or a running application, passes through the API server which keeps the cluster state stored in `etcd`, a distributed key-value database. The scheduler determines on which worker node each new pod should run, while the controller manager executes the control loops that continuously compare the current state of the cluster with the desired one and take the necessary actions to keep them aligned.

Worker nodes are responsible for running application workloads, also with the support of the kubelet, which communicates with the API server and ensures that the pods assigned to the node are running correctly. The actual execution of containers is handled by a container runtime, while `kube-proxy` manages the networking rules that allow traffic to be correctly routed to the appropriate pods.

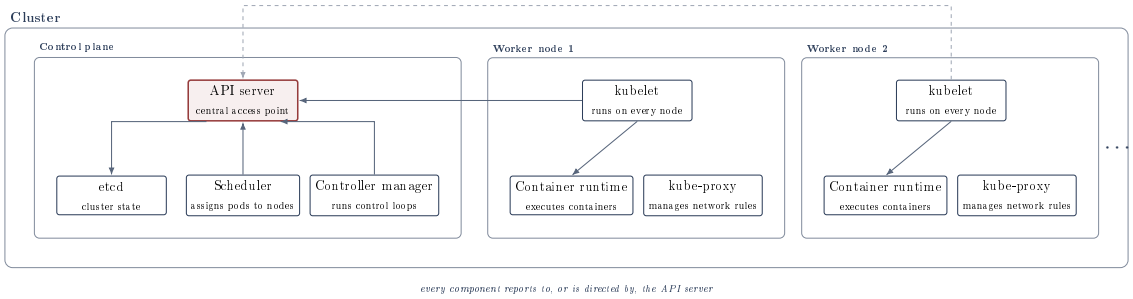


Figure 2.1: The control plane makes decisions about the cluster and stores its state while every worker node runs the components needed to execute the workloads the control plane assigns to it.

2.2 Core objects

Kubernetes organises applications through a small set of core objects, each with a specific role in the management of the cluster:

1. The **pod** is the smallest deployable unit in Kubernetes and it consists of one or more containers that share the same network identity, including a single IP address and can also share storage resources. In practice, pods are rarely created manually, as usually, higher-level objects, such as Deployments, are responsible for creating and managing them, automatically replacing failed pods and gradually rolling out new versions of an application.

2. A **Service** provides a stable name and a virtual IP address for a group of pods selected through labels and this allows clients to communicate with the application without depending on the individual addresses of pods, which may change over time as pods are created or replaced.

3. A **namespace** is a logical partition of the cluster that is used to organise and isolate Kubernetes resources. Other objects such as Pods, Services and policies may exist within a namespace. Most Kubernetes resources are namespaced, although some of them, such as nodes and cluster-wide roles, exist at the cluster level. This grouping is independent of node placement as pods in the same namespace can run on different nodes and a single node can run pods from several namespaces.

4. A **ServiceAccount** is the identity used when interacting with the API server that every pod is associated with and it is linked with a token that is mounted inside the pod itself, allowing processes running in its containers to authenticate to the API server.

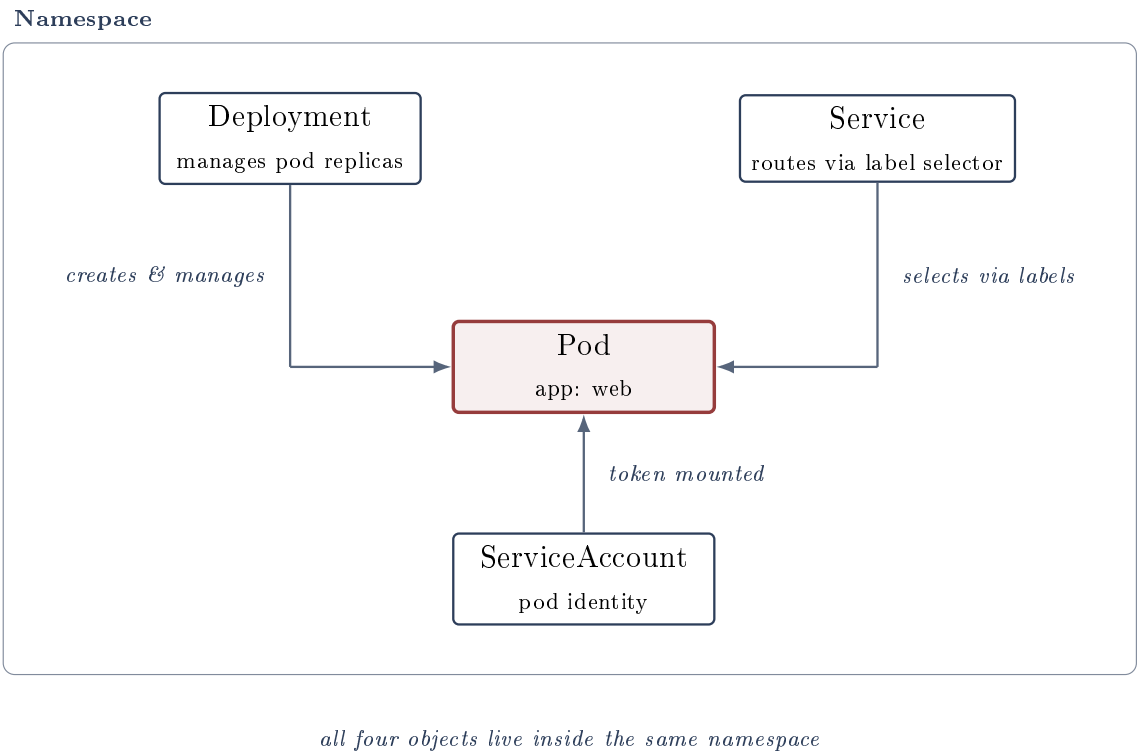


Figure 2.2: A Deployment creates and manages the Pods matching its template. A Service reaches those same Pods dynamically, through a label selector rather than a fixed address and the Pod authenticates to the API server with the token of the ServiceAccount mounted inside it.

2.3 The two security mechanisms

A request sent to the Kubernetes API server goes through three stages:

- **authentication**, which verifies the identity of the caller;
- **authorisation**, which decides whether the requested operation is allowed;
- **admission**, where further validation or modification is applied before the object is stored.

Traffic between pods is handled separately, by the networking layer and never passes through the API server.

Among the security mechanisms Kubernetes offers, two carry most of the weight and are the focus of this thesis.

RBAC (Role-Based Access Control) [56] is responsible for authorisation and defines through `Role`, or its cluster-wide counterpart `ClusterRole` a list of rules.

Each rule names three things: the verbs it allows (get, list, create, delete and so on), the resources those verbs apply to (pods, secrets, networkpolicies and the rest) and the API groups those resources belong to. A rule grants nothing on its own because it takes effect only once a `RoleBinding` or a `ClusterRoleBinding` ties it to a subject which can be a user, a group, or a ServiceAccount. RBAC therefore decides which actions each identity may perform through the API.

NetworkPolicy [53] controls the network communication between pods by selecting a group of pods through their labels and specifies which ingress and egress traffic is allowed. Once a pod is selected by a policy for a given direction, any traffic that is not explicitly permitted in that direction is denied. If no NetworkPolicy applies, the default behaviour is to allow communication between all pods in the cluster. A NetworkPolicy is a declarative object and is not enforced by the API server. Moreover, it takes effect only if the cluster runs a **Container Network Interface (CNI)** plugin that implements enforcement, such as Calico or Cilium. A cluster can therefore hold NetworkPolicy objects that look correct and are never applied.

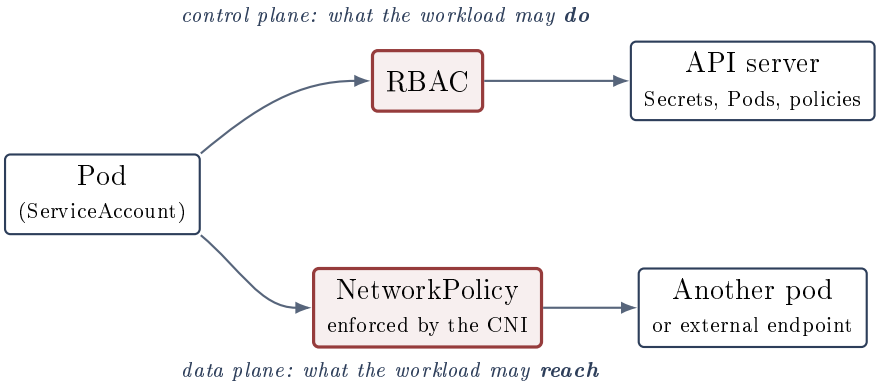


Figure 2.3: RBAC and NetworkPolicy protect two different paths through different mechanisms which never interfere with each other

2.4 Why these two mechanisms

RBAC is evaluated by the API server and controls operations performed on the control plane, whereas NetworkPolicy is enforced by the CNI and regulates network communication on the data plane. Together, these mechanisms define both what a workload is allowed to **do** and which resources it is allowed to **reach**. Since they rely on different resources and are enforced by different components, they are often configured and analysed independently.

However, considering them separately can hide security issues that only become visible when both mechanisms are evaluated together. A weakness in RBAC may increase the impact of a weak NetworkPolicy configuration and vice versa. For this reason, analysing only one of the two mechanisms does not always provide a complete view of the cluster's security posture.

Figure 2.3 illustrates the different responsibilities of RBAC and NetworkPolicy.

Chapter 3

Motivation

Chapter 2 introduced the two mechanisms that a Kubernetes cluster relies on to protect itself: RBAC, evaluated by the API server, decides what each identity may do, and NetworkPolicy, enforced by the CNI, decides which pods it may reach. Both are essential and both are usually configured and inspected on their own. This chapter explains why looking at them in isolation leaves a gap that neither side can see.

3.1 Problem statement

Misconfigurations in each mechanism are well studied in isolation. In fact, detection tools such as Kubescape [3], Trivy [2], KICS [14] and Checkov [5] each scan one domain and the published guidance (best-practice cheat-sheets [45], CIS benchmarks [1], the CVE record [19]) almost always treats RBAC and NetworkPolicy as separate concerns.

What exists therefore describes each domain on its own and rarely the interaction between an RBAC weakness and a NetworkPolicy one, even though real attacks routinely chain the two. An overprivileged ServiceAccount is far more dangerous in a flat, unisolated network, because a single compromised pod can then reach every other workload. The reverse holds too as a missing egress restriction matters far more when the identity behind it can already read every Secret in the cluster. The danger of a single misconfiguration is therefore not always a fixed property of that object, since it also depends on the configuration of the other plane, which no single domain tool inspects.

An audit can therefore declare a cluster healthy in each domain separately while the combination is still exploitable. The CVE system catalogues software defects rather than configuration combinations, so the most consequential cross-domain failures leave almost no trace in the public vulnerability record as nobody builds tooling for them and nobody measures them.

3.2 Thesis objective

This is the gap this thesis addresses. This thesis builds a taxonomy of Kubernetes misconfigurations that covers RBAC and NetworkPolicy on their own and then the layer where the two interact. The patterns are derived bottom up, starting from

what the security community already detects, so each one corresponds to a check performed by real tooling and, where possible, to a documented CVE. RBAC and NetworkPolicy share the same collection method and the same organising principle, since families in both are grouped by the security property that breaks. This keeps the two taxonomies comparable and lets them be composed into a third.

Chapter 4

Methodology and Dataset

This chapter explains the methodology and principles used to build the taxonomies. It is deliberately a single chapter covering both sub-projects, RBAC (Chapter 5) and NetworkPolicy (Chapter 6), as they are collected with the same method. This is what allows the two taxonomies to be compared directly.

4.1 Organising principles

Two principles shape the whole work and are applied identically to both domains. The first is **organisation by the nature of the failure**: families (L1) group misconfigurations by which security property is violated and not by the object that carries the defect, the tool that happens to flag it, or the size of the impact. Each family answers a single specific question: "Which security guarantee does this misconfiguration break?" as this criterion was chosen because it is intrinsic to the defect, unlike the object or the tool, which are respectively syntax and collection metadata. It also gives every misconfiguration exactly one home, so nothing has to be filed twice and it yields the same set of buckets in both domains, which is what makes the RBAC and NetworkPolicy taxonomies directly comparable. Within a family, patterns (L2) and subpatterns (L3) separate the concrete mechanism that produces the failure. A pattern is subdivided only where its mechanism admits distinct variants, for example eight of the thirty-two patterns do not and their L3 column carries a dash.

The second principle is **empirical grounding**: every pattern corresponds to something real, either a check that detection tooling already looks for or, where the surveyed tools do not cover it, a mechanism grounded in official documentation, a security advisory or a CVE instead. Together they keep the taxonomy reproducible and comparable across the two domains. When a pattern is not covered by any of the surveyed tools, it is marked as analytical. This does not mean that the pattern is based only on our own reasoning as most of these cases are still supported by external documentation, security advisories, or other references. The fact that a pattern is not currently checked by the surveyed tools does not mean that the underlying weakness is not real. It only means that it is not covered by the tools considered in this work, which is why we keep it in the taxonomy rather than excluding it.

Pattern format. Every pattern in the three taxonomies is presented in the same fixed sequence, so the catalogue also works as a practical reference:

1. **Definition** of the mechanism.
2. **Threat narrative**, with its MITRE ATT&CK mapping.
3. **Detection criterion** (the predicate a tool or reviewer checks).
4. **CVE evidence**, where a verified case exists.
5. **Collected entries**, the corpus support for the pattern.
6. **L3 variants**, each with a vulnerable manifest and, where the fix takes the form of a manifest, a corrected policy.
7. **Remediation**, usually shared once across every L3 variant of the pattern, though a few variants whose fix differs get one of their own.

Keeping this sequence identical across RBAC, NetworkPolicy and the cross-domain taxonomy is what lets the catalogue also work as a practical reference.

Classification dimensions. Each entry is placed along three explicit axes, so that it is always clear why a misconfiguration sits where it does.

- **Nature of the failure (L1):** the security property that breaks. For example, an identity receives power it should not, or a pod is reachable when it should be unreachable.
- **Mechanism (L2/L3):** the concrete mechanism that produces the misconfiguration. It could be a wildcard rule, an empty selector, a missing `policyTypes` and so on.
- **Impact scope:** what the misconfiguration affects, recorded per entry during review. It can be a namespace (the effect stays within one namespace) or cluster (it extends cluster-wide). A family's **effect** line reports its worst-case ImpactScope. If a family spans both levels, this is stated. Moreover, an escalation to a wider reach through a later step (for example, from a reachable metadata endpoint to node credentials) is described in prose.

A note on the ATT&CK version. Each family and cross-pattern is mapped to MITRE ATT&CK [18]. Since ATT&CK is updated over time, technique names and identifiers may change between releases. The mappings used in this thesis refer to **ATT&CK Enterprise v19.1**, which became current in April 2026. When a technique is part of the Containers platform, this is stated explicitly. Some mappings refer instead to techniques that are not specific to Containers. For example, ATT&CK does not provide a Containers-specific technique for exfiltration over a command-and-control channel, so the general technique is used in those cases. Readers using a different ATT&CK release may therefore find different technique names or identifiers. In particular, v19 retired `T1562` "Impair Defenses" and replaced it with the new Defense Impairment tactic and Disable or Modify Tools (`T1685`).

The name of each cross-family names the overall effect it produces, not a claim that every technique listed under it carries that exact tactic in ATT&CK. A table can pair a family with a technique that supports the effect at an earlier step, such

as credential exposure that later leads to exfiltration, without that technique itself being filed under the family's own tactic. Where a step has no dedicated technique at all, such as the bare reachability that becomes access in `X7`, the mapping stops at the closest one and says so rather than forcing a fit.

One family per misconfiguration. Each family guards one property and collects the ways that property fails (Table 4.1).

	Family	The property it guards
RBAC	**R-F1** Identity Overexposure	an identity holds only the privileges it needs
	R-F2 Permission Overreach	a rule grants only what it needs
	R-F3 Sensitive Resource Exposure	sensitive resources and the records the cluster keeps are touched only by the subjects that need them
	R-F4 Privilege Escalation Paths	no primitive allows privilege escalation
	R-F5 Authorization Layer Preconditions	RBAC is actually enforced
NetworkPolicy	**N-F1** Missing Isolation	a pod is isolated by an explicit policy
	N-F2 Over-permissive Connectivity	an existing policy allows only the peers and ports needed
	N-F3 Ineffective Policy	a policy that exists actually works
	N-F4 Sensitive Endpoint Exposure	sensitive endpoints stay fenced off
	N-F5 Enforcement Layer Preconditions	the CNI enforces the policies
	N-F6 CNI-extension & Admin Policy	admin and cluster-wide policies do not override intended denials

Table 4.1: Security property each family guards: a misconfiguration belongs to the family whose property it breaks, which is what gives every entry exactly one home.

How entries were assigned. The families were not fixed in advance. They emerged bottom up from the collected detection rules (Section 4.2), which is what gives the taxonomy its coverage of the space tools actually check. Each collected entry was then placed by identifying which property it breaks. Where a defect breaks more than one (a rule can be both broad and about a sensitive resource) the properties are tested in a fixed order, most fundamental first. The first match sets the family and the order is stated below:

#	Test applied, in order	Family
RBAC		
1	Enforcement of RBAC is itself disabled or bypassed (a control plane setting)	**R-F5**
2	The defect is a binding to the wrong or over-powerful subject	**R-F1**
3	A rule grants an escalation primitive (`bind`/`escalate`/`impersonate`, pod create or exec, webhook, CSR approval, token minting, node access)	**R-F4**
4	A rule targets a sensitive resource (`secrets`, `configmaps`, `events`, `pods/log`)	**R-F3**
5	Anything else is raw breadth (wildcards, manage-all)	**R-F2**
NetworkPolicy		
1	The CNI does not enforce, or the pod leaves the pod network	**N-F5**
2	The object is an admin or cluster-wide policy (ANP/BANP, Cilium or Calico cluster-wide)	**N-F6**
3	No policy selects the pod	**N-F1**
4	A policy selects it but does nothing	**N-F3**
5	An active policy leaves a sensitive endpoint reachable (metadata, kubelet, etcd, database)	**N-F4**
6	Anything else is general over-permissiveness (broad peer, `ipBlock`, ports)	**N-F2**

Table 4.2: How an entry is assigned to a family: the tests are applied in this order and the first match decides, which is what gives every entry exactly one home.

The fixed order means the same entry always lands in the same family. Where a defect also enables another one, the overlap is recorded on the pattern rather than filed twice.

Figure 4.1 puts the two results side by side. Both have the same shape, with a first level of families named after the property that breaks, then a second and a third level for the mechanism. The last family on each side is the borderline one, which collects conditions of the substrate rather than of the configuration. It is this shared shape that later makes it meaningful to pair an RBAC family with a NetworkPolicy family.

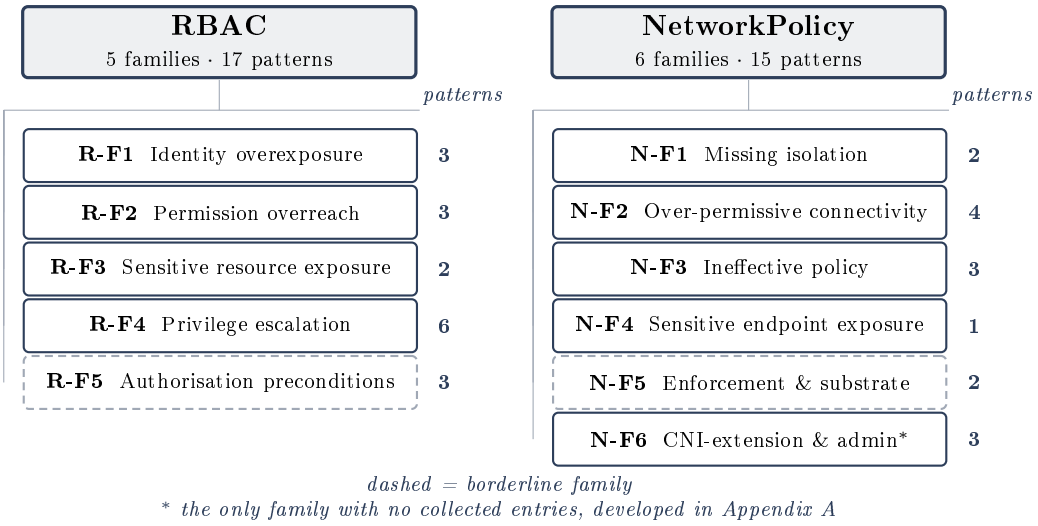


Figure 4.1: The two single-domain taxonomies share one shape:families named after the property that breaks, patterns for the mechanism, subpatterns for the variants. The parallel structure is what makes the two directly comparable and what allows a family from one side to be composed with a family from the other.

4.2 The collection method

Collection is performed by two scripts, `rbac_collector.py` and `netpol_collector_v2.py`, which share the same skeleton. For each source they retrieve the detection rules directly from the upstream GitHub repository, reduce them to a common entry schema, classify each one, de-duplicate and enrich with CVEs. Running either script reproduces its dataset from scratch. Figure 4.2 shows the whole path from a rule in an external repository to a pattern in this taxonomy. The first four steps are automatic and repeatable while the review is manual and it is where the numbers drop the most, because a script can tell that an entry mentions NetworkPolicy but not whether it is really about pod security or a deprecated API.

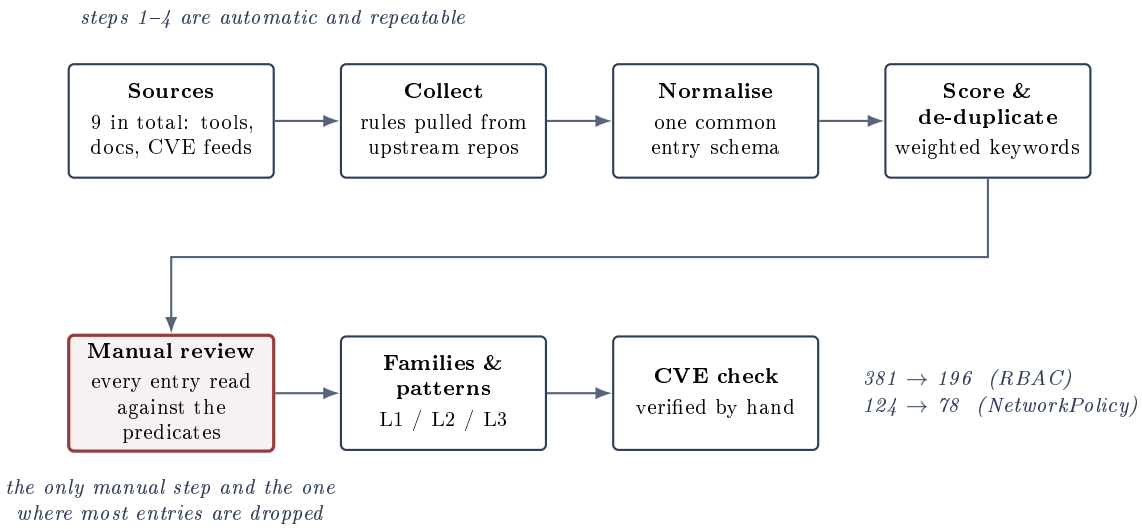


Figure 4.2: From a detection rule in a public repository to a pattern in the taxonomy: the scoring step decides whether an entry concerns the domain at all and the manual review decides whether it survives and accounts for most of the reduction across the two datasets.

Normalisation

Every collected item (a Rego rule, a Kyverno policy, a CIS benchmark line, a documentation paragraph) is reduced to a common entry, so that the two domains can be analysed with the same tooling. The schema has two halves (Table 4.3): the fields the collector fills in on its own and the fields added by hand during the review. The second half is where the numbers reported in Chapter 5 and Chapter 6 come from.

Field	Meaning
Filled by the collector	
`source`	which tool or document the entry comes from
`title`, `description`	the rule as its author wrote it
`url`	where it can be read upstream
`keywords`, `categories`	the terms that fired during classification
`score`, `*_type`	the classification score and the resulting label
`cve_count`, `max_cvss`	CVE enrichment from the NVD and CVE List feeds
Added during the manual review	
`Inclusion`	kept or discarded and on what grounds
`Family`, `Pattern`	where the entry lands in the taxonomy
`DuplicateOf`	the entry this one restates, if any
`ImpactScope`	namespace or cluster
`Notes`	why the call was made, when it was not obvious

Table 4.3: The entry schema: the upper half is produced automatically and is reproducible by re-running the collector while the lower half is the reviewer's and carries the decisions the taxonomy rests on.

What an entry looks like

The word "entry" recurs throughout this chapter, so it is worth showing one. Table 4.4 follows entry #38 from the RBAC corpus through the whole pipeline. It starts as a Kubescape control, a paragraph of English prose with an identifier and ends as a row that the taxonomy can count.

Two strong categories have been linked on this entry, core and escalation. The domain term also appears in the title, where matches count three times compared to the text. The score of 134 therefore clears the STRICT threshold by a wide margin. The script cannot decide on its own whether the defect is an identity holding powers it should not have rather than a rule that is too broad. That judgement is the reviewer's and it is what puts the entry in **R-F1** rather than **R-F2**.

As published — the Kubescape control	
`controlID`	`C-0185`
`name`	Ensure that the cluster-admin role is only used where required
`description`	"The RBAC role `cluster-admin` provides wide-ranging powers over the environment and should be used only where and when needed."
`remediation`	Identify every subject bound to `cluster-admin` and confirm the access level is actually required; where it is not, rebind to a narrower role and delete the binding.
As collected — filled by the script	
`source`	`kubescape_control`
`title`	C-0185 — Ensure that the cluster-admin role is only used where required
`description`	"The RBAC role `cluster-admin` provides wide-ranging powers over the environment and should be used only where and when needed."
`url`	`https://hub.armosec.io/docs/c-0185`
`keywords`	cluster-admin, clusterrole, clusterrolebinding, rolebinding, rbac, admin role
`categories`	core, escalation
`score`	134
`rbac_type`	STRICT
After the review — added by hand	
`Inclusion`	Included
`Family`	**R-F1** Identity Overexposure
`Pattern`	`R-P1.1` Cluster-admin binding
`ImpactScope`	cluster
`Component`	`ClusterRoleBinding`
`Notes`	same defect as the CIS 5.1.1 line already recorded, kept as a separate detector rather than a duplicate

Table 4.4: One entry, end to end: the Kubescape control as published, what the collector stores from it, and the fields the manual review adds. It appears as #38 in the entry table of `R-P1.1`.

Classification: the central decision

The hard part is deciding which collected items actually concern the domain and how strongly. The scripts use a weighted scoring classifier with three refinements:

- **Weighted keywords, title-boosted:** each keyword carries a weight and a category. For example a match in the title counts three times, since a title term is far more indicative than an incidental body mention.
- **Context-aware negatives:** negative terms subtract points only when the text contains no hard domain term, so an entry that merely mentions an unrelated concept by analogy is not penalised, while an entry that is genuinely off-topic is correctly demoted.
- **Three-way labelling by score:** items are labelled STRICT (strong anchor, high score), RELATED (lower score), or SECURITY_POSTURE (no strong anchor but security-posture categories). Everything else is discarded.

These thresholds separate the strict core from the adjacent and posture material, which is the distinction each taxonomy later makes explicit.

Table 4.5 reports the actual scoring scheme: each keyword category carries a weight and a flag stating whether it is a strong anchor (a strong category must appear at least once for an item to qualify as STRICT/RELATED). Title matches count three times and the security posture categories (sensitive, components, indirect, compliance) are the ones that can promote an otherwise weak item to SECURITY_POSTURE.

Table 4.5: The classification scoring scheme (RBAC collector).

Category	Weight	Strong?	Role / example terms
escalation	6	yes	escalate, bind, impersonate
dangerous_api	6	yes	pods/exec, certificatesigningrequests
core	5	yes	rbac, role, clusterrole, binding
misconfig	4	yes	wildcard, overly permissive
rbac_subject	4	yes	serviceaccount, group, user
sensitive	3	no	secret, token (posture)
managed_k8s_iam	3	no	IRSA, workload identity
indirect	2	no	configmap, webhook (posture)
components	1	no	apiserver, kubelet (posture)
compliance	1	no	CIS, benchmark (posture)
platforms/workloads	0	no	context only, no score

Modifiers: title match $\times 3$/bonus pair (for example rbac+kubernetes) $+3$/negative term -5 only if no hard RBAC anchor.
Labels: strong $\wedge$ score $\geq 25 \rightarrow$ STRICT/$\geq 10 \rightarrow$ RELATED/posture $\wedge$ score $\geq 8 \rightarrow$ SECURITY_POSTURE/otherwise discarded.

The full collectors (`rbac_collector.py`, `netpol_collector.py`) are provided in the appendix for full reproducibility. This chapter covers only the decision logic, not the underlying infrastructure

CVE enrichment

Each retained entry is matched against the NVD API [19] and the CVE List v5 feed [15] through an automation in the script. When a match is found, the corresponding CVEs, together with their CVSS scores and severity ratings, are recorded for that entry. For NetworkPolicy, matching needs an extra safeguard. In fact, CVEs for Calico, Cilium or Antrea often describe bugs unrelated to policy enforcement, so matching on the product name alone would link a pattern to CVEs that have nothing to do with it. An early version did exactly that and produced 699 links from only 45 unique CVEs, most of them indefensible. The process now first keeps only CVEs whose description explicitly mentions NetworkPolicy enforcement or isolation, then links each entry only to CVEs that share a specific NetworkPolicy keyword with it, and keeps at most the ten strongest matches per entry, ranked by shared keywords and CVSS. Each NetworkPolicy entry additionally includes a computed `risk_score` that combines these signals into a single ranking value.

Limitation: CVE attribution works at the level of the whole dataset, not always for each individual pattern. CVE enrichment links CVEs to detection tool entries by checking which keywords occur together. This is useful to show that the domain is tied to real, often severe vulnerabilities (216 RBAC entries match at least one CVE). However, it is too coarse to attribute CVEs to individual patterns, because a small number of generic Kubernetes CVEs with high severity appear across many different patterns. For this reason, associations between a CVE and a specific pattern are reported only where they have been manually verified against the official CVE description (for example `CVE-2019-11247` [23] for wildcard verbs, `CVE-2021-25740` [25] for aggregated role abuse, `CVE-2023-2728` [28] for Secret access). The remaining patterns rely only on the evidence already described above. Table 4.6 summarises how many CVEs were manually verified for each family, and how many of them are quoted directly in a pattern box rather than recorded in Appendix B only.

Family	Verified CVEs	Quoted in text
R-F1	3	3
R-F2	2	2
R-F3	2	2
R-F4	8	0
R-F5	1	0
Cross-domain[†]	1	1
N-F2	2	2
N-F5	3	3
N-F6	2	2
Total	**24**	**15**

Table 4.6: Manually verified CVEs by family, and how many are quoted directly in a pattern box rather than recorded in Appendix B only.

[†] `CVE-2024-31989` is not mapped to any RBAC family; it is quoted in the cross-domain chapter (Chapter 7) instead.

4.3 Sources

Source selection is an integral part of the methodology. In order to derive patterns from how the community detects Kubernetes misconfigurations, collection must cover the full detection spectrum, from preventive admission policies (Kyverno [17], Gatekeeper [44]) to static CI/CD scanners (Trivy [2], KICS [14], Checkov [5]), runtime behavioural detection (Falco [50]), compliance benchmarks (CIS via kube-bench [1]) and authoritative first-party documentation (Kubernetes documentation [56, 53], OWASP [45]).

Since no single tool covers all classes of Kubernetes misconfigurations, a multi-source corpus is required. The RBAC and NetworkPolicy domains share a core set of eight sources and then diverge, since NetworkPolicy required domain-specific tools for misconfigurations that existing public tooling does not cover.

4.3.1 RBAC sources

For RBAC, nine sources are used (Table 4.7). They were selected to cover the main stages of the Kubernetes security ecosystem. Unlike the other sources, the ninth is not a detection tool but the CVE feeds (NVD and CVE List v5), which are used to associate known vulnerabilities and CVSS scores with each collected entry.

Table 4.7: RBAC: nine collection sources.

Source	**Contribution**
Kubescape (controls + rules) [3]	runtime / posture detection logic
Trivy (trivy-checks) [2]	static CI/CD scanning rules (Rego)
Falco (rules) [50]	behavioural runtime detection
Gatekeeper (library) [44]	OPA admission policy templates
kube-bench (CIS) [1]	CIS Kubernetes compliance benchmarks
Kyverno (policies) [17]	policy-as-code enforcement rules
OWASP K8s cheatsheet [45]	community best-practice guidance
Kubernetes documentation [56, 53]	authoritative API semantics
CVE feeds (NVD, cvelistV5) [19, 15]	vulnerability and CVSS metadata

4.3.2 NetworkPolicy sources

NetworkPolicy reuses the shared RBAC sources and adds the domain-specific sources listed in Table 4.8. A few relevant misconfigurations are not covered by any surveyed public tool such as allow-all rules through empty peers, unrestricted `ipBlock 0.0.0.0/0` without exceptions, incorrect selector composition, overly permissive port ranges and cluster-wide CNI or administrative overrides. These are kept as **analytical** patterns.

Table 4.8: NetworkPolicy: additional sources on top of the shared RBAC dataset.

Added source	**Contribution**
KICS (Checkmarx) [14]	static IaC scanning of NetworkPolicy manifests
Checkov (Bridgecrew) [5]	policy-as-code checks for Kubernetes manifests
network-policy API (SIG) [16]	AdminNetworkPolicy / baseline specification
NetworkPolicy recipes [4]	canonical community examples
kube-linter [49]	lint rules for weak or missing policies

4.4 The two datasets

The RBAC corpus consists of **381 entries**, reduced through manual review to **196 retained entries**. The NetworkPolicy collection consists of **124 entries**, reduced to **78 retained entries**.

The two classification axes are independent. Strict versus related versus security-posture describes what an entry is about, while retention describes whether the review kept it. Table 4.9 summarises both datasets. The per-family breakdown for each domain is given at the head of Chapter 5 and Chapter 6.

Table 4.9: Summary of the two datasets (collected → retained).

	RBAC	**NetworkPolicy**
Collected entries	381	124
strict	206	75
related	45	18
security-posture	130	31
with ≥ 1 CVE match	216	79
Retained after review	**196**	**78**
Excluded / unclassified	185	45

4.5 Scope and domain distinctions

The strict/borderline distinction has slightly different meanings in the two domains and is therefore described separately. In RBAC, strict entries are fully contained within RBAC objects (bindings and rules), while borderline entries extend to adjacent mechanisms such as PodSpec configuration, admission controllers or external identity systems (for example, default ServiceAccount cases in `R-P1.3`). In NetworkPolicy, strict entries are fully expressible within NetworkPolicy objects while borderline entries instead concern enforcement semantics, CNI behaviour, or administrative policies at the cluster level.

These borderline cases are included because they capture security issues that go beyond the core object model, but are still relevant for cluster security. They are also particularly important for the cross-domain analysis in Chapter 7.

Two domain-specific adaptations. Both domains use the same weighted scoring model, the same STRICT (≥ 25) and RELATED (≥ 10) thresholds and the same three-class classification scheme in order to preserve comparability.

The differences concern two domain-specific aspects.

- The SECURITY_POSTURE threshold is set to 8 for RBAC and 15 for NetworkPolicy, because NetworkPolicy contains a higher concentration of posture entries with low variance, which needed stronger filtering
- Each NetworkPolicy entry also includes a computed `risk_score` and `risk_rank` while RBAC entries include `attack_tracks` and `control_tags` instead. These are representations specific to each domain of the same underlying scoring model, not methodological deviations.

Chapter 5

RBAC Misconfigurations

This chapter presents the first taxonomy, focusing on RBAC misconfigurations derived from the corpus described in Chapter 4. Role-Based Access Control (RBAC) is the native authorisation mechanism provided by Kubernetes, implemented through the rbac.authorization.k8s.io API group and operating under a deny-by-default model: a subject is granted access only if an explicit rule permits it. The model is built around four core API objects (Role, ClusterRole, RoleBinding, and ClusterRoleBinding) which together define what actions are allowed (verbs on specific resources and API groups) and to whom they are granted (users, groups, or service accounts). Roles and ClusterRoles scope permissions to a single namespace or the entire cluster, respectively, while RoleBindings and ClusterRoleBindings link those permissions to subjects. Because permissions are additive across all applicable bindings, even a single overly permissive rule can silently break the least-privilege guarantee and open a path to full cluster compromise. Recent work has compared RBAC with alternative access-control models for the same class of platforms [48]. The misconfigurations identified in this chapter are organised into five families, each representing a distinct security dimension, and further split into patterns (L2) and subpatterns (L3):

R-F1 — **Identity Overexposure** Misconfigurations that assign permissions to identities that should not receive them, typically through overly broad bindings involving users, groups, or ServiceAccounts.

R-F2 — **Permission Overreach** Misconfigurations that grant permissions beyond what is required, including wildcard rules, overly permissive roles, and misuse of aggregated roles.

R-F3 — **Sensitive Resource Exposure** Misconfigurations that expose access to sensitive Kubernetes resources such as Secrets, configuration objects, or observability-related data.

R-F4 — **Privilege Escalation Paths** Misconfigurations that enable subjects to obtain higher privileges through capabilities such as `bind`, `escalate`, `impersonate`, or through pod creation and execution permissions.

R-F5 — **Authorization Layer Preconditions** Misconfigurations affecting the Kubernetes authorization layer itself, including disabled authorizers or other configuration issues that weaken RBAC enforcement.

5.1 Corpus statistics

The RBAC corpus comprises **381** collected entries (see Chapter 4). After manual review, **196** entries are retained and mapped to the five families defined above.

Of these, **154** entries are assigned to one of the five families, while **42** entries are retained as security-posture context. These latter cases represent borderline or adjacent material that does not map cleanly to a single family but is preserved for completeness, following the same rationale introduced in Chapter 4.

Bucket	Entries	Meaning
Total collected	381	raw detection entries (Chapter 4)
R-F1 Identity Overexposure	43	mapped entries
R-F2 Permission Overreach	14	mapped entries
R-F3 Sensitive Resource Exposure	28	mapped entries
R-F4 Privilege Escalation Paths	40	mapped entries
R-F5 Authorization Layer Preconditions	29	mapped entries
Security-posture residual	42	contextual (no family assignment)
Retained after review	**196**	154 family-mapped + 42 posture

Table 5.1: Data funnel for the RBAC domain: reduction from 381 collected entries to 196 retained entries.

5.2 Family R-F1 — Identity Overexposure

Forty-three entries are assigned to this family (Included + Borderline), spread across three patterns.

Effect. A compromised overprivileged identity acts with permissions far beyond its role, up to full cluster-admin (ImpactScope: cluster).

Family definition

Family **R-F1** collects the misconfigurations in which an identity (a named user, a group, or a ServiceAccount) ends up with more power over the cluster than its legitimate role requires. In most cases the mechanism is the RBAC binding (`RoleBinding` or `ClusterRoleBinding`), which decides how much power an identity carries. `R-P1.3.2` is the exception, since it lives in the PodSpec and decides who can pick up that identity. The two form two halves of the same failure as a binding that no workload can reach exposes nothing a mounted token is what turns code execution inside a container into an authenticated client of the API server.

Threat narrative

An attacker who compromises an overprivileged **R-F1** identity gets an attack surface far wider than the application's legitimate role needs. The usual vector is the placement of the wrong subject inside an RBAC binding but as said above `R-P1.3.2` is

the exception, where the credential itself is the defect. The best-documented case is the Tesla 2018 incident in which a Kubernetes Dashboard exposed via a public LoadBalancer with a `ClusterRoleBinding` to `system:authenticated` turned every authenticated request into a cluster-admin and was abused for cryptocurrency mining. **R-F1** is often the starting point of attack chains that continue with **R-F4** (escalation) and **R-F3** (credential theft).

MITRE ATT&CK mapping. In the MITRE ATT&CK framework (the industry-standard catalogue of observed adversary techniques) these scenarios map to two techniques.

Technique	Why it applies
Valid Accounts `T1078`	the binding hands the adversary those credentials directly, no exploit needed
Account Manipulation: Additional Container Cluster Roles `T1098.006`	the adversary creates or edits a binding to gain that privilege

General detection criterion

Detection criterion — family R-F1

There exists an identity s such that either (i) an RBAC binding b grants s, through b.`roleRef`, a permission set strictly larger than the least-privilege set required by s's legitimate function, or (ii) s is a control plane credential attached to a workload that has no legitimate need for it.

Patterns of family R-F1

Pattern	Name	L3 variants	Entries
`R-P1.1`	Cluster-admin binding	`R-P1.1.1` ClusterRoleBinding to cluster-admin `R-P1.1.2` RoleBinding to cluster-admin	8
`R-P1.2`	System groups binding	`R-P1.2.1` Anonymous `R-P1.2.2` Authenticated `R-P1.2.3` Masters	16
`R-P1.3`	Privileged SA distribution	`R-P1.3.1` Admin Overprivilege `R-P1.3.2` Default SA Exposure	19

5.2.1 R-P1.1 — Cluster-admin binding

Pattern definition. A subpattern of **R-F1** in which a `(Cluster)RoleBinding` has the built-in ClusterRole `cluster-admin` as its `roleRef`. The bound subject then obtains unlimited power over the cluster (all verbs on all resources of all apiGroups) through the role-binding mechanism.

Boundary with `R-P1.2.3`. The special group `system:masters` produces the same outcome (unconditional super user) but through a different mechanism which is through group membership bypassing RBAC at the API-server level, not a `roleRef` to a named role, so it is tracked separately as `R-P1.2.3` rather than folded into this predicate. Overall the two patterns are equivalent in effect but distinct in mechanism.

Detection criterion — R-P1.1

A binding's `roleRef` points to the built-in ClusterRole `cluster-admin`. *Formally:* $\exists b :$ b.`roleRef` = `cluster-admin`. (For the group-subject mechanism, see `R-P1.2.3`.)

Collected entries. Eight entries, all with ImpactScope = Cluster:

Entries	Detection (source)
#32, #38, #77, #107	cluster-admin role / binding detection (Kubescape [3])
#59, #144	cluster-admin restrictive-use checks (Trivy [2])
#4	restrict binding to cluster-admin (Kyverno [17])
#42	CIS 5.1.1 — minimise cluster-admin use (CIS)

Entry #144 (Trivy `authenticate_group_bind.rego`) actually targets `system:authenticated` bindings (`R-P1.2.2`) but it is kept here as a Trivy-side variant of the broader "restrict cluster-admin/system-group use" family of checks.

L3 — R-P1.1.1 ClusterRoleBinding to cluster-admin

A `ClusterRoleBinding` whose `roleRef` is `cluster-admin` has cluster-wide effect on all present and future namespaces. This can be considered the most general and severe form of the pattern.

✗ **Vulnerable manifest** (triggers R-P1.1.1)

```yaml
# ClusterRoleBinding tying an application ServiceAccount to cluster-admin:
# the subject gains unlimited power over the ENTIRE cluster.
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: app-backup-binding
subjects:
  - kind: ServiceAccount
    name: backup-sa
    namespace: backups
roleRef:
  kind: ClusterRole
  name: cluster-admin           # <-- triggers predicate R-P1.1
  apiGroup: rbac.authorization.k8s.io
```

L3 — R-P1.1.2 RoleBinding to cluster-admin

A namespace-scoped `RoleBinding` that references the `cluster-admin` ClusterRole. Apparently milder (single-namespace scope), but it makes the subject a **namespace admin** which can allow to read the namespace's Secrets and, because it can create new bindings within the namespace, can also pivot to other RBAC roles. Only the binding kind changes from `R-P1.1.1`.

✗ Vulnerable manifest (triggers R-P1.1.2)

```yaml
# Namespace-scoped variant: same cluster-admin roleRef, but a RoleBinding.
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: dev-admin-binding
  namespace: dev
roleRef:
  kind: ClusterRole
  name: cluster-admin          # <-- de facto admin of namespace "dev"
  apiGroup: rbac.authorization.k8s.io
subjects:
  - kind: ServiceAccount
    name: dev-sa
    namespace: dev
```

Remediation. Subpatterns `R-P1.1.1` and `R-P1.1.2` share one remediation, addressed on three levels.

- **Structural fix**: replace the role by swapping `cluster-admin` for a service-specific ClusterRole that enumerates only the needed verbs (for example, a backup operator needs `get/list/watch` on all resources and no `create/update/delete`).
- **Operational controls**: block the binding at admission time with the Kyverno policy of entry #4 (Restrict Binding to Cluster-Admin) in `Enforce` mode and audit the live cluster regularly, since the admission policy only stops new bindings and cannot catch ones already present.
- **Temporary permissions**: when someone genuinely needs cluster-admin during an incident, the binding should carry an expiry rather than be created by hand. Kyverno can attach a time to live annotation to the binding and delete it once the time is up. Most of the permanent cluster-admin bindings found in real clusters were created for an emergency that ended long ago.

Audit / detection

```
# Detection 1 - list every binding to cluster-admin (kubectl + jq):
kubectl get clusterrolebindings -o json \
  | jq '.items[] | select(.roleRef.name=="cluster-admin") | .metadata.name'

# Detection 2 - Kubescape control (entry #77):
kubescape scan control rule-list-all-cluster-admins-v1
```

Enforcement policy

```yaml
# Kyverno (entry #4): deny, at admission, any binding to cluster-admin.
apiVersion: kyverno.io/v1
kind: ClusterPolicy
metadata:
  name: restrict-binding-clusteradmin
spec:
  validationFailureAction: Enforce
  rules:
    - name: block-clusteradmin-binding
      match:
        any:
          - resources:
              kinds: ["ClusterRoleBinding", "RoleBinding"]
      validate:
        message: "Direct binding to cluster-admin is not allowed."
        deny:
          conditions:
```

```
            any:
              - key: "{{ request.object.roleRef.name }}"
                operator: Equals
                value: "cluster-admin"
```

5.2.2 R-P1.2 — System groups binding

Pattern definition. A subpattern of **R-F1** in which a `Role` or `ClusterRole` is assigned, through a `RoleBinding` or `ClusterRoleBinding`, to one of the built-in Kubernetes system identities. These are not identities an administrator creates but the API server assigns them on its own based on how a request authenticates, so a binding to one of them applies to a population rather than to a subject.

- `system:anonymous` is the username the API server assigns to a request that arrives with no credentials at all. A binding to it grants permissions to anyone who can reach the API server endpoint.
- `system:unauthenticated` is the group that those same requests belong to. It is listed separately because a binding can name either the user or the group and a check that looks for only one of the two misses half the cases.
- `system:authenticated` contains every identity that has authenticated successfully through client certificates, tokens, OIDC and every ServiceAccount in the cluster. It is therefore far broader than the human users of the cluster and a permission granted here also reaches every workload that carries a token.
- `system:masters` is hardcoded in the API server. The RBAC authorizer allows any request from a member of this group without consulting a single rule. The bound role is irrelevant and no RBAC policy can restrict it. Membership normally comes from a client certificate whose organisation field is `system:masters`, which is how the administrative `kubeconfig` produced by most installers is built.

The pattern is further divided into three L3 subpatterns according to the target system identity.

Detection criterion — R-P1.2

A binding matches this pattern if one of its subjects is a `Group` named `system:anonymous`, `system:unauthenticated`, `system:authenticated` or `system:masters` (or the `User` `system:anonymous`).

Collected entries. Sixteen entries (15 Included, 1 Borderline), all with ImpactScope = Cluster. They divide across the three L3 as 4 anonymous, 7 authenticated and 3 masters, plus two preventive checks that intercept several system groups at once:

Entries	Detection (source)
#22, #225	`system:authenticated` holding elevated roles (Kubescape)
#58, #158	anonymous access enabled / anonymous RoleBinding (Kubescape)
#94, #161, #187	Dashboard reachable through a broad binding (Kubescape)
#235	C-0246 — avoid use of `system:masters` (Kubescape)
#133, #143	`system:masters` and anonymous bindings (Trivy)
#247	CIS 5.1.7 — avoid use of the `system:masters` group (CIS)
#23, #54	preventive: restrict binding to system groups (Kyverno)
#65	self-provisioner binding on OpenShift (Kyverno)
#245	disallow anonymous subjects at admission (Gatekeeper)
#135	guideline: securing the Dashboard (OWASP)

L3 — R-P1.2.1 Anonymous

Binding of `system:anonymous`/`system:unauthenticated` grants permissions to requests without any credential, so anyone who can reach the API server becomes authorised. This can be considered the most severe form of exposure, as it requires no credential, no prior knowledge of secrets, and no stolen token but only network reachability to the API server to gain authorisation.

✗ Vulnerable manifest (triggers R-P1.2.1)

```yaml
# Grants a role to unauthenticated requests: no credential required at all.
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: anonymous-read
roleRef:
  kind: ClusterRole
  name: view
  apiGroup: rbac.authorization.k8s.io
subjects:
  - kind: Group
    name: system:anonymous        # <-- triggers predicate R-P1.2
    apiGroup: rbac.authorization.k8s.io
```

L3 — R-P1.2.2 Authenticated

Binding of `system:authenticated` reaches a group that includes every authenticated identity regardless of method, not just human users but every ServiceAccount in the cluster, so a role that looks scoped to a handful of operators is in practice reachable by any pod that carries a token. The example below binds the Dashboard's ClusterRole this way, turning what looks like a narrow grant into cluster-wide exposure.

✗ VULNERABLE MANIFEST (triggers R-P1.2.2)

```yaml
# Binds the ENTIRE authenticated population to a broad ClusterRole:
# any authenticated client inherits these permissions.
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: dashboard-binding
roleRef:
  kind: ClusterRole
  name: kubernetes-dashboard      # overly broad role
  apiGroup: rbac.authorization.k8s.io
subjects:
  - kind: Group
    name: system:authenticated    # <-- triggers predicate R-P1.2
    apiGroup: rbac.authorization.k8s.io
```

L3 — R-P1.2.3 Masters

An identity that authenticates as a member of `system:masters` (a group hard-coded in the API server, most commonly reached through a client certificate whose `Subject` carries that organisation) bypasses RBAC entirely and is an unconditional super-user, regardless of any role bound to it. A `(Cluster)RoleBinding` naming `system:masters` as a subject, shown below, does not grant that membership, it can only bind additional, redundant permissions to whoever already holds it, so the binding is a symptom worth auditing for, not the mechanism itself. The two patterns, `R-P1.1` and `R-P1.2.3`, are detected independently because they are triggered by different fields: a `roleRef` to a named role for the first, evidence of `system:masters` membership for the second.

✗ VULNERABLE MANIFEST (triggers R-P1.2.3)

```yaml
# A subject already authenticated as a member of system:masters bypasses
# RBAC and is an unconditional super-user, regardless of the bound role;
# membership itself comes from authentication (e.g. a client certificate
# with O=system:masters), not from this binding. The binding below is a
# symptom worth auditing for, since it signals that membership was assumed.
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: masters-binding
roleRef:
  kind: ClusterRole
  name: view                      # role is irrelevant: the group bypasses RBAC
  apiGroup: rbac.authorization.k8s.io
subjects:
  - kind: Group
    name: system:masters          # <-- signals assumed system:masters
        membership
    apiGroup: rbac.authorization.k8s.io
```

Remediation. Subpatterns `R-P1.2.1`–`R-P1.2.3` share the same remediation strategy, which can be applied at two levels:

- **Structural level** (covered by **R-F5**): set `-anonymous-auth=false` on both the API server and the kubelet, which makes anonymous bindings ineffective.
- **RBAC level**: enforce a Gatekeeper (#245) or Kyverno policy that denies any `RoleBinding` or `ClusterRoleBinding` targeting `system:unauthenticated`, `system:authenticated` or `system:masters` directly.

As an exception, `system:anonymous` should remain bound only to `system:public-info-viewer`, allowing public health endpoints such as `/healthz` and `/livez` to remain accessible for load balancer health checks.

Enforcement policy

```
# Kyverno: deny bindings to system groups (except anonymous -> public-info-
    viewer).
apiVersion: kyverno.io/v1
kind: ClusterPolicy
metadata:
  name: restrict-system-group-binding
spec:
  validationFailureAction: Enforce
  rules:
    - name: deny-anonymous-binding
      match:
        any:
          - resources: { kinds: ["ClusterRoleBinding", "RoleBinding"] }
      validate:
        message: "system:anonymous may only be bound to system:public-info-
            viewer."
        deny:
          conditions:
            all:
              - key: "{{ request.object.subjects[?name=='system:anonymous'] |
                  length(@) }}"
                operator: GreaterThan
                value: 0
              - key: "{{ request.object.roleRef.name }}"
                operator: NotEquals
                value: "system:public-info-viewer"
    - name: deny-other-system-groups
      match:
        any:
          - resources: { kinds: ["ClusterRoleBinding", "RoleBinding"] }
      validate:
        message: "Binding to system groups is not allowed."
        foreach:
          - list: "request.object.subjects"
            deny:
              conditions:
                any:
                  - key: "{{ element.name }}"
                    operator: AnyIn
                    value: ["system:unauthenticated",
                            "system:authenticated", "system:masters"]
```

5.2.3 R-P1.3 — Privileged SA distribution

Pattern definition. A subpattern of **R-F1**, explicitly classified as MIXED-DOMAIN, describing misconfigurations related to the assignment and use of `ServiceAccount` privileges within a Kubernetes cluster. Unlike `R-P1.1` and `R-P1.2`, which are entirely defined by RBAC bindings, this pattern also involves adjacent security mechanisms, including Pod specifications, cross-platform identity integration and admission plugins. For this reason, the pattern is divided into two L3 subpatterns:

- `R-P1.3.1` Admin Overprivilege: covering excessive privileges assigned to ServiceAccounts through RBAC.
- `R-P1.3.2` Default ServiceAccount Exposure: a workload that uses the namespace `default` ServiceAccount and/or mounts an API token it does not need.

Detection criterion — R-P1.3

The pattern is matched when either (i) a ServiceAccount is bound, through an RBAC binding, to a role granting at least one cluster-takeover capability while the binding does not name `cluster-admin` directly (`R-P1.3.1`), or (ii) a Pod carries a ServiceAccount token it has no legitimate use for, because `serviceAccountName` is left unset or because `automountServiceAccountToken` is not disabled on the pod or the account (`R-P1.3.2`). The first condition is read from the RBAC graph and the second from the PodSpec, which is why the pattern spans two adjacent mechanisms instead of one.

Real-world evidence (CVE)

Manually verified, core-Kubernetes cases where ServiceAccount credentials leak or are reused: `CVE-2019-11243` [22] (`AnonymousClientConfig` returned a config still carrying the bearer token), and `CVE-2025-7445` [40] / `CVE-2023-2878` [30] (`secrets-store-sync-controller` / `secrets-store-csi-driver` disclosing SA tokens in logs). None of the three is an instance of `R-P1.3`'s own detection criterion, an over-privileged binding or an unnecessarily mounted token: `CVE-2019-11243` is a client-library sanitisation bug that failed to strip a token from a config object, and the two logging CVEs leak a token that already sits on its intended workload through a controller's own log output. Neither path runs through the RBAC graph or the PodSpec. All three still anchor the pattern because the leaked material is the same ServiceAccount token `R-P1.3` is about, and a token read from a client config or a log is exactly as usable to an attacker as one read from a Pod that should never have carried it. Evidence and severity anchor for `R-P1.3`.

Collected entries. The most populated pattern of the family and the one most skewed towards Borderline, which reflects its mixed-domain nature:

Entries	Detection (source)
#39, #44, #127, #231	workload whose SA carries administrative or cluster-takeover roles (Kubescape)
#99, #171	container access to the ServiceAccount token (Kubescape)
#82, #136	automatic mapping and automount of the default SA token (Kubescape)
#84, #129	default SA left with roles, namespace without a dedicated SA (Kubescape)
#164, #180	dedicated SAs on managed platforms, AKS and ECR (Kubescape)
#125	CIS 5.1.5 — default SAs should not be actively used (CIS)
#146, #151, #204	`ServiceAccount` admission plugin enabled (Kubescape, Trivy, CIS 1.2.13)
#60, #61	SA-token Secret type denied at admission (Kyverno)
#98	cloud identity federation for the CNI DaemonSet, IRSA (Kyverno)

L3 — R-P1.3.1 Admin Overprivilege

A workload whose ServiceAccount is bound to a Role/ClusterRole with cluster-takeover capabilities. A misconfiguration defining a deliberate excess of power relative to the workload's minimum privilege.

Detection criterion — R-P1.3.1

A binding whose subject is a ServiceAccount and whose role grants at least one "cluster-takeover" capability (for example, `create pods`, `create rolebindings`, `escalate roles`, `impersonate *`, `* secrets`, `* serviceaccounts`) other than a literal `roleRef` to `cluster-admin`, which is `R-P1.1` regardless of subject kind.

✗ Vulnerable manifest (triggers R-P1.3.1)

```yaml
# An application's SA bound to a ClusterRole carrying takeover powers.
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: app-sa-takeover
roleRef:
  kind: ClusterRole
  name: secret-and-pod-manager     # includes *, secrets and create, pods
  apiGroup: rbac.authorization.k8s.io
subjects:
  - kind: ServiceAccount
    name: app-sa
    namespace: prod
```

L3 — R-P1.3.2 Default SA Exposure

`R-P1.3.1` is about an identity that can do more than it should while `R-P1.3.2` is about a credential the workload never asked for. What happen is that pod leaves `serviceAccountName` unset, so it runs as the namespace `default` ServiceAccount and it mounts that account's API token even though it never talks to the API server. Every pod is assigned a ServiceAccount, so the identity itself is unavoidable but what is avoidable is sharing it with every other pod in the namespace and keeping its token inside the container. The defect holds even when the account carries no permissions at all, because it sits in the PodSpec and not in an RBAC rule. It still falls under **R-F1**, since the family's detection criterion already treats an unnecessary credential attached to a workload as one of the two conditions that define overexposure, independently of what that credential is authorized to do.

Detection criterion — R-P1.3.2

The pattern is matched when a pod either (i) runs under the namespace `default` ServiceAccount, no `serviceAccountName` being set, or (ii) mounts a ServiceAccount token, `automountServiceAccountToken` being not disabled on either the pod or the account, while making no use of the API server. The first two conditions are read from the manifest while whether the workload genuinely needs the credential is the judgement added during the manual review.

✗ Vulnerable manifest (triggers R-P1.3.2)

```yaml
# A pod that mounts the namespace "default" SA token without needing it:
# any code execution in the pod can reuse that token against the API server.
apiVersion: v1
```

```yaml
kind: Pod
metadata:
  name: web
  namespace: prod
spec:
  # serviceAccountName omitted  -> uses "default"
  automountServiceAccountToken: true   # <-- token mounted unnecessarily
  containers:
    - name: web
      image: nginx:1.27
```

Remediation. The two subpatterns share the same goal, giving each workload exactly the ServiceAccount and the permissions it needs, but they address it from two different angles.

- **RBAC side**: Kubescape #39 (`workload-with-cluster-takeover-roles`) flags the overprivileged binding, while the actual minimum scope can be derived by enabling audit logging at the `RequestResponse` level for one to two weeks. The resulting verb×resource combinations feed `audit2rbac`, which generates a minimal `ClusterRole` automatically. Workloads that legitimately need elevated privileges, for example cert-manager, Falco [50] and Argo CD, are not stripped of them but contained instead through a dedicated namespace, a default-deny `NetworkPolicy`, immutable image digests and continuous auditing of runtime behaviour.
- **Credential side**: the default should be `automountServiceAccountToken: false` on the ServiceAccount or the pod, with each workload receiving a dedicated minimal ServiceAccount instead of a shared one and the namespace `default` SA left with no permissions at all.

Note (cloud identity federation). A related but distinct case is cloud-to-Kubernetes identity federation (for example, IRSA on EKS, Workload Identity on GKE), where a pod's ServiceAccount is mapped to a cloud IAM role. It shares the same root (an identity a pod carries without needing it) but lives at the cloud-IAM boundary rather than in the PodSpec and is noted here only for completeness.

5.3 Family R-F2 — Permission Overreach

This family has 14 entries, all Included and with ImpactScope = Cluster in all but one entry. ***R-F2*** *is entirely strict-RBAC.*

Effect. A single over-broad rule exposes a large slice of the API surface and if combined with escalation verbs it reaches full cluster compromise (ImpactScope: cluster).

Family definition

Family **R-F2** covers RBAC misconfigurations that are not about who holds the permission, but about how far the permission itself reaches. In these cases the binding to the identity can be entirely correct and the excess sits in the rule itself, which ends up granting more than the operational role actually needs, regardless of which of its three patterns is at fault.

Threat narrative

A subject bound to an over-permissive rule in **R-F2** does not need to escalate or to change identity, because the permissions it already holds cover a large part of the API surface. How large that part is depends on how many verbs and resources the rule reaches.

This family rarely does the damage on its own and in fact it is what makes another weakness worth exploiting. A wildcard rule that also carries escalation verbs opens the way to full cluster compromise, and a manage-all rule becomes a workload takeover as soon as one ServiceAccount is compromised. Aggregation is the most difficult to diagnose of the three, because it can extend privileges across namespaces without anyone editing a role (`R-P2.3`).

MITRE ATT&CK mapping. In MITRE ATT&CK these scenarios map to the following techniques.

Technique	Why it applies
Valid Accounts `T1078`	the rule already grants the adversary those permissions
Account Manipulation: Additional Container Cluster Roles `T1098.006`	the excess permissions can modify RBAC objects themselves

General detection criterion

Detection criterion — family R-F2

There exists an RBAC rule whose capability set, given by the combination of verbs, resources and API groups, is strictly larger than the one required by the operational role the rule is meant to serve. The defect is in the rule itself, whatever subject a binding attaches to it.

Patterns of family R-F2

Pattern	Name	L3 variants	Entries
`R-P2.1`	Wildcard permissions	`R-P2.1.1` Wildcard in verbs `R-P2.1.2` Wildcard in resources `R-P2.1.3` Wildcard in API groups	8
`R-P2.2`	Manage-all over a scope	`R-P2.2.1` Cluster-wide manage-all `R-P2.2.2` Namespace-wide manage-all `R-P2.2.3` Single-API-group manage-all	3
`R-P2.3`	Aggregated ClusterRole abuse	`R-P2.3.1` Aggregation into a built-in role `R-P2.3.2` Aggregation into a custom aggregate role	3

5.3.1 R-P2.1 — Wildcard permissions

Pattern definition. A subpattern of **R-F2** in which a rule uses the wildcard `*` in one or more of the fields `verbs`, `resources`, or `apiGroups`. The wildcard expands the match set to all possible values of the field in which it appears, often leading to unintended broad access. When combined across multiple fields, the effect approaches a full cluster level grant and overlaps with `R-P2.2`. The pattern remains strictly within RBAC, as the issue is entirely contained in rule specification.

Detection criterion — R-P2.1

A rule contains `*` in at least one of `verbs`, `resources`, or `apiGroups`.

Real-world evidence (CVE)

A validated reference illustrating the risk of this pattern is `CVE-2019-11247` [23]. In this vulnerability, a defect in the API server allowed a request targeting a cluster-scoped custom resource to be evaluated as if it were namespaced, so a namespace-level Role naming the matching resource type authorised access to the cluster-scoped object too. A wildcard rule such as `R-P2.2.1` is one way to name that resource, since it matches whatever the request asks for, but the official advisory does not limit the exploitable rules to wildcards, since any namespace Role naming the resource works the same way. `R-P2.1` is still the broadest and most certain path to the bug, which is why the CVE anchors it here rather than at `R-P2.2.1` alone.

Collected entries. Eight entries, all with ImpactScope = Cluster, distributed across the L3 variants by the field carrying the wildcard. A fully open rule combining multiple wildcards is classified under `R-P2.2` to avoid double counting.

Entries	Detection (source)
#88, #90	restrict wildcards in `resources` (Kyverno)
#89, #91	restrict wildcards in `verbs` (Kyverno)
#153, #241	any verb, any verb on any resource (Trivy)
#167	C-0187 — minimise wildcard use in Roles and ClusterRoles (Kubescape)
#170	CIS 5.1.3 — minimise wildcard use (CIS)

L3 — R-P2.1.1 Wildcard in verbs

A rule that allows all verbs on a fixed set of resources, so the resource type is scoped correctly but every action on it is left open. A subject bound to it can read, create, update or delete every object of that type, which for a resource such as Pods is already enough to run arbitrary code in the cluster.

✗ **Vulnerable manifest** (triggers R-P2.1.1)

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: pod-anything
rules:
- apiGroups: [""]
  resources: ["pods"]
```

```
  verbs: ["*"]
```

L3 — R-P2.1.2 Wildcard in resources

A rule that allows specific actions on all resources within an API group, so the verbs stay limited but the resource type is unrestricted. Even the read-only verbs `get`, `list` and `watch` reach every resource in the group when combined this way, Secrets and ConfigMaps included, which is enough to expose cluster-wide credentials without any write access.

✗ Vulnerable manifest (triggers R-P2.1.2)

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: read-anything
rules:
- apiGroups: [""]
  resources: ["*"]
  verbs: ["get", "list", "watch"]
```

L3 — R-P2.1.3 Wildcard in API groups

A rule that applies across all API groups, so a verb and resource combination meant for one group reaches every group that happens to define a resource with the same name. The example lists Deployments in every API group instead of just `apps`, which silently pulls in any custom resource that reuses the name.

✗ Vulnerable manifest (triggers R-P2.1.3)

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: list-across-groups
rules:
- apiGroups: ["*"]
  resources: ["deployments"]
  verbs: ["list"]
```

Remediation. All wildcard-based rules should be replaced with explicitly enumerated verbs, resources, and apiGroups. Admission-time enforcement is implemented via the Kyverno [17] policy below, matching entries #88 and #89, which rejects any rule containing `*` in verbs, resources, or apiGroups.

Enforcement policy

```yaml
# Kyverno (entries #88, #89): deny a Role or ClusterRole using a wildcard
# in verbs, resources, or apiGroups.
apiVersion: kyverno.io/v1
kind: ClusterPolicy
metadata:
  name: restrict-wildcard-rbac
spec:
  validationFailureAction: Enforce
  rules:
    - name: no-wildcard-in-rules
      match:
        any:
          - resources:
              kinds: ["Role", "ClusterRole"]
      validate:
```

```
      message: "Wildcards (*) are not allowed in verbs, resources, or
          apiGroups."
      foreach:
        - list: "request.object.rules"
          deny:
            conditions:
              any:
                - key: "{{ element.verbs[] }}"
                  operator: AnyIn
                  value: ["*"]
                - key: "{{ element.resources[] }}"
                  operator: AnyIn
                  value: ["*"]
                - key: "{{ element.apiGroups[] }}"
                  operator: AnyIn
                  value: ["*"]
```

5.3.2 R-P2.2 — Manage-all over a scope

Pattern definition. `R-P2.2` rules grant full control over a scope by combining `verbs=['*']` and `resources=['*']`. The scope can be the whole cluster, a single namespace, or one API group, such as `networking.k8s.io` or `storage.k8s.io`. `R-P2.1` only reaches part of the API through wildcards but this pattern goes further, because the rule ends up fully permissive within its scope and at broad scope it is essentially a superuser role.

Detection criterion — R-P2.2

A rule contains `*` in both `verbs` and `resources` at the same time. The scope it reaches, whether the whole cluster, a single namespace or one API group, follows from whether the rule sits in a `ClusterRole` or a `Role` and from whether `apiGroups` is itself left as `*` or narrowed to one named group.

Collected entries. Three entries, one per L3 and all from the same tool. Only Trivy encodes the manage-all shape as a check of its own.

Entries	Detection (source)
#152	manage all resources cluster-wide (Trivy) — `R-P2.2.1`
#236	manage all resources within a namespace (Trivy) — `R-P2.2.2`
#208	manage all networking resources (Trivy) — `R-P2.2.3`

L3 — R-P2.2.1 Cluster-wide manage-all

Full control, `verbs=['*']` and `resources=['*']`, granted through a `ClusterRole` with no group or namespace restriction. Anyone bound to it can create, read, update, or delete any object in the cluster, RBAC rules included, which makes the role equivalent to `cluster-admin` even though it is never named that way.

✗ **Vulnerable manifest** (triggers R-P2.2.1)

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: superpowers
```

```
rules:
- apiGroups: ["*"]
  resources: ["*"]
  verbs: ["*"]
```

L3 — R-P2.2.2 Namespace-wide manage-all

The same shape, `verbs=['*']` and `resources=['*']`, but scoped to a single namespace through a `Role` instead of a `ClusterRole`. The reach stays inside that namespace, but there the holder can touch every resource, Secrets, ConfigMaps and RoleBindings included and can grant itself further access by editing the bindings directly.

✗ Vulnerable manifest (triggers R-P2.2.2)

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: ns-superpowers
  namespace: team-a
rules:
- apiGroups: ["*"]
  resources: ["*"]
  verbs: ["*"]
```

L3 — R-P2.2.3 Single-API-group manage-all

Full control (`verbs=['*']`, `resources=['*']`) scoped to one `apiGroup`. The example grants every verb on every object in `networking.k8s.io`. The holder can create, edit, or delete all NetworkPolicies, Ingresses and IngressClasses in the cluster, enough to tear down network isolation or reroute ingress traffic, even though the role names no wildcard `apiGroup`. It is more confined than `R-P2.2.1`, but still a superuser inside that domain.

✗ Vulnerable manifest (triggers R-P2.2.3)

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: networking-admin
rules:
- apiGroups: ["networking.k8s.io"]
  resources: ["*"]
  verbs: ["*"]
```

Remediation. Rules of this form should not exist in production systems. When they do appear, they must be replaced with narrowly scoped roles that explicitly enumerate the required operations. Furthermore, because broad scope control can also manifest through exhaustive explicit enumerations of critical primitives rather than pure wildcards, administrative roles and bindings must be audited holistically. Admission controllers should also block any rule combining full wildcards on both verbs and resources, enforced at admission time with the Kyverno policy below.

Enforcement policy

```
# Kyverno: deny a Role or ClusterRole whose rule combines a wildcard in
# both verbs and resources at the same time.
apiVersion: kyverno.io/v1
kind: ClusterPolicy
```

```yaml
metadata:
  name: restrict-manage-all-rbac
spec:
  validationFailureAction: Enforce
  rules:
    - name: no-verbs-and-resources-wildcard-together
      match:
        any:
          - resources:
              kinds: ["Role", "ClusterRole"]
      validate:
        message: "A rule may not combine a wildcard in verbs and resources at
            the same time."
        foreach:
          - list: "request.object.rules"
            deny:
              conditions:
                all:
                  - key: "{{ element.verbs[] }}"
                    operator: AnyIn
                    value: ["*"]
                  - key: "{{ element.resources[] }}"
                    operator: AnyIn
                    value: ["*"]
```

5.3.3 R-P2.3 — Aggregated ClusterRole abuse

Pattern definition. Aggregation is a legitimate Kubernetes feature where A ClusterRole with an `aggregationRule` automatically unions every ClusterRole matching its label selector. It becomes a risk when a role carrying sensitive permissions is labelled so that it is pulled into a broader aggregate. Two variants follow as descrived below:

Detection criterion — R-P2.3

A ClusterRole carries a label matching the `clusterRoleSelectors` of an `aggregationRule`, either (i) on a built-in role (`view`, `edit` or `admin`), which is `R-P2.3.1`, or (ii) on a custom aggregate, which is `R-P2.3.2`, while its own `rules` grant a sensitive permission the aggregate target was not meant to carry. The defect is not in the labelled ClusterRole taken alone but in what the label injects into the role it aggregates into, which only a check that follows the aggregation label can find.

Real-world evidence (CVE)

`CVE-2021-25740` [25] serves as a documented instance reflecting this risk category. In this vulnerability, permissions associated with built-in roles (such as `edit`) allowed subjects to modify `Endpoint` or `EndpointSlice` objects to point to internal addresses across namespaces, effectively escalating ordinary resource-modification rights into a cluster-wide traffic interception vector.

Collected entries. Three entries, all in `R-P2.3.1` and all built around the same CVE, which shows how narrowly the tools read this mechanism. `R-P2.3.2` generalises it to custom aggregates and is stated on analytical grounds, since no surveyed tool checks it.

Entries	Detection (source)
#63, #64	restrict `edit` on Endpoints (Kyverno)
#116	block Endpoint edit in the default role (Gatekeeper)

L3 — R-P2.3.1 Aggregation into a built-in role

The aggregation target is one of the built-in roles, `view`, `edit`, or `admin`, which Kubernetes ships with the `rbac.authorization.k8s.io/aggregate-to-*` labels already wired in. Any ClusterRole carrying the matching label is pulled into the built-in role automatically, so labelling a role with elevated permissions is enough to grant them to everyone who already holds `edit`or `admin`, with no explicit binding required.

✗ **Vulnerable manifest** (triggers R-P2.3.1)

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: extra-endpoints
  labels:
    rbac.authorization.k8s.io/aggregate-to-edit: "true"
rules:
- apiGroups: [""]
  resources: ["endpoints"]
  verbs: ["create", "update", "patch"]
```

L3 — R-P2.3.2 Aggregation into a custom aggregate role

The same mechanism, but the target is an aggregate role defined by the organisation rather than a built-in one. Any actor able to create or label a ClusterRole with the aggregate's selector label injects its permissions into that aggregate, so a low-privilege subject with control over ClusterRole labels can escalate indirectly. No surveyed tool generalises the check beyond the built-in case, so the subpattern is stated on analytical grounds.

✗ **Vulnerable manifest** (triggers R-P2.3.2)

```yaml
# A custom aggregate: it unions every ClusterRole carrying the selector label.
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: company-admin
aggregationRule:
  clusterRoleSelectors:
    - matchLabels: { company.example.com/aggregate: "true" }
rules: []             # filled automatically from matching ClusterRoles
---
# Labelling any ClusterRole with that key injects its rules into company-admin:
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: sneaky-extra
  labels: { company.example.com/aggregate: "true" }
rules:
  - apiGroups: [""]
    resources: ["secrets"]
    verbs: ["*"]      # <-- silently added to company-admin (R-P2.3.2)
```

Remediation. A misconfiguration of this caliber needs to be fixed with a combination of technical and organisational measures, such as the following:

- **Upgrade**: move to Kubernetes 1.22 or later, where sensitive permissions were removed from the default aggregated roles.
- **Legacy clusters**: restrict aggregation to roles that do not touch sensitive resources and add admission-time validation on the aggregation labels.
- **Custom aggregates**: treat the selector label used for `R-P2.3.2` as privileged, with tight control over who can create or label ClusterRoles carrying it.

5.4 Family R-F3 — Sensitive Resource Exposure

*Twenty-eight entries are assigned to this family with half of them being borderline. **R-F3** is the most mixed domain family of the taxonomy.*

Effect. Disclosure of credentials and configuration and loss of auditability may have serious consequences, with escalation risk that depends on the resource (namespaced for a `Role`, cluster-wide for a `ClusterRole`). (ImpactScope: namespace or cluster).

Family definition

Family **R-F3** groups the misconfigurations in which a rule names a resource the subject should not reach, namely the credentials the cluster stores, the configuration it runs on or the record it keeps of what happened.

The affected resources include `Secret` (credentials, tokens, certificates and keys), `ConfigMap` (application and system configuration, sometimes containing cleartext credentials), `Events` (cluster-level audit information) and `pods/log` (runtime logs that may contain sensitive data). Even minimal permissions such as `get` or `list` are sufficient to cause meaningful information disclosure when applied to these resources.

Threat narrative

This family enables credential exposure, configuration tampering and loss of auditability. In this scenario the attacker already has some execution capability in a workload and uses an overly permissive RBAC rule or a misconfigured default to reach Secrets or configuration objects. From there the credentials are extracted or the system's behaviour is changed.

A representative case is CoreDNS modification via ConfigMap access in the `kube-system` namespace. CoreDNS configuration lives in a ConfigMap, so write access lets an attacker change DNS behaviour and redirect internal service traffic. Access to Events or logs opens up a different kind of impact in the anti-forensics field as it may let an attacker erase traces of prior actions after compromise.

MITRE ATT&CK mapping. In MITRE ATT&CK terms, the family spans multiple techniques depending on the resource.

Technique	Why it applies
Unsecured Credentials: Container API T1552.007	grants direct read access to credentials stored in Secrets
Stored Data Manipulation T1565.001	grants write access to configuration such as CoreDNS
Indicator Removal T1070	lets the attacker delete logs or Events to erase traces

General detection criterion

Detection criterion — family R-F3

An RBAC rule grants any verb over a resource classified as sensitive (`secrets`, `configmaps`, `events`, or `pods/log`) to a subject that does not require access to that data for its intended function.

Patterns of family R-F3

Pattern	Name	L3 variants	Entries
R-P3.1	Secret access	R-P3.1.1 Secret read access R-P3.1.2 Secret write access	20
R-P3.2	Config and observability access	R-P3.2.1 ConfigMap write R-P3.2.2 ConfigMap read R-P3.2.3 Events deletion R-P3.2.4 Pod log deletion	8

5.4.1 R-P3.1 — Secret access

Pattern definition. This pattern captures RBAC rules that grant access to Kubernetes Secrets, where even read only access can result in credential disclosure. Two L3 subpatterns separate read access (R-P3.1.1) from write access (R-P3.1.2). How a Secret is consumed once read (environment variable vs mounted file) and how it is stored (encryption at rest, external secret stores) are adjacent concerns rather than RBAC rules and they are collected in the note below.

Detection criterion — R-P3.1

An RBAC rule includes `secrets` (or a named Secret through `resourceNames`) among its resources, under either (i) the read verbs `get`, `list` or `watch` (R-P3.1.1), or (ii) the write verbs `create`, `update`, `patch` or `delete` (R-P3.1.2), granted to a subject with no operational need for that credential material.

Real-world evidence (CVE)

Relevant CVEs include CVE-2023-2728 [28] and CVE-2024-3177 [37], where a bypass of the `ServiceAccount` admission plugin's mountable-secrets policy let a

workload mount a Secret its own ServiceAccount was never bound to. Neither CVE is an instance of `R-P3.1`'s own detection criterion, an RBAC rule granting read or write access to Secrets, since the gap they exploit sits in admission control rather than in an authorization rule. They still provide empirical grounding for the pattern because the resulting exposure is the same one an over-broad `R-P3.1` rule would produce: a workload reading Secret material it has no operational need for.

Collected entries. Twenty entries, the most populated pattern of the taxonomy after `R-P5.1`. Half of them are direct rules on the `secrets` resource, the rest are Borderline and concern how Secrets are stored or consumed, which is where the family becomes mixed domain.

Entries	Detection (source)
#50, #52	restrict Secret verbs in Roles (Kyverno)
#62, #66	disallow rules covering all Secrets (Kyverno)
#78, #92	manage secrets, cluster-wide and namespaced (Trivy)
#138, #166, #172	`get`/`list` access to Secrets (Kubescape)
#209	CIS 5.1.2 — minimise access to secrets (CIS)
#306	C-0255 — workload with Secret access (Kubescape)
#267, #274	Secrets exposed as environment variables (CIS 5.4.1, Kubescape)
#278, #300, #366	external secret storage (Kubescape, CIS 5.4.2)
#315	C-0244 — Secrets encrypted at rest (Kubescape)
#345	C-0012 — credentials in configuration files (Kubescape)
#327, #328	ingress snippets able to read Secrets (Kyverno)

L3 — R-P3.1.1 Secret read access

Read access to Secrets through verbs such as `get`, `list` and `watch` is sufficient to expose all credentials stored in a namespace.

✗ Vulnerable manifest (triggers R-P3.1.1)

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: secret-reader
  namespace: prod
rules:
- apiGroups: [""]
  resources: ["secrets"]
  verbs: ["get", "list", "watch"]
```

L3 — R-P3.1.2 Secret write access

Write access to Secrets allows creation or modification of credentials, effectively enabling impersonation or persistence mechanisms.

✗ **Vulnerable manifest** (triggers R-P3.1.2)

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: secret-manager
rules:
- apiGroups: [""]
  resources: ["secrets"]
  verbs: ["create", "update", "patch", "delete"]
```

Note — Secret consumption and storage Seven of the entries above are not RBAC rules at all and they are kept because they complete the picture of how a Secret gets exposed. Two hygiene concerns border the pattern from the workload and the platform side. The first one speaks about how a Secret is **consumed**. In fact, injecting it as an environment variable (`env` or `envFrom` with `secretKeyRef`) leaks far more easily than mounting it as a file, because it surfaces in the process environment, in crash dumps, in `kubectl describe` output and in any child process. The exposure is therefore independent of whether the RBAC rules are correct and standard tooling checks it (CIS 5.4.1, Kubescape [3], Kyverno [17]).

The second is how a Secret is **stored** as encryption at rest and external secret stores are mitigating controls that live on the platform rather than in an RBAC rule. Neither concern changes what a rule grants, which is why they are noted here instead of being given a subpattern of their own.

Remediation.

- **Access policy**: Secrets access should follow strict least-privilege principles, with explicit justification required for any read or write grant, enforced at admission time with the Kyverno [17] policy below, which matches entries #50 and #52 and denies the write verbs on Secrets for any Role or ClusterRole outside the platform's own.
- **Workload level**: Secrets should be consumed via mounted volumes rather than environment variables.
- **Platform level**: encryption at rest and external secret management systems reduce exposure in case of RBAC misconfiguration.

Enforcement policy

```yaml
# Kyverno (entries #50, #52): deny a Role or ClusterRole granting write
# verbs on Secrets.
apiVersion: kyverno.io/v1
kind: ClusterPolicy
metadata:
  name: restrict-secret-write-verbs
spec:
  validationFailureAction: Enforce
  rules:
    - name: no-secret-write-verbs
      match:
        any:
          - resources:
              kinds: ["Role", "ClusterRole"]
      validate:
        message: "Secrets may only be granted the read verbs get, list and watch
            ."
        foreach:
          - list: "request.object.rules"
```

```
            deny:
              conditions:
                all:
                  - key: "{{ element.resources[] }}"
                    operator: AnyIn
                    value: ["secrets"]
                  - key: "{{ element.verbs[] }}"
                    operator: AnyIn
                    value: ["create", "update", "patch", "delete"]
```

5.4.2 R-P3.2 — Config and observability access

Pattern definition. This pattern covers RBAC rules that grant access to configuration and observability resources, specifically `ConfigMap`, `Events`, and `pods/log`. These resources do not inherently contain credentials, but they may expose sensitive operational information, permit changes to system behaviour, or affect the visibility of security-relevant activity. The subpatterns therefore distinguish between information disclosure, configuration tampering, and loss of operational or runtime evidence.

Detection criterion — R-P3.2

An RBAC rule grants either (i) `update` or `patch` on `configmaps` (`R-P3.2.1`), (ii) `get` or `list` on `configmaps` (`R-P3.2.2`), (iii) `delete` or `deletecollection` on `events` (`R-P3.2.3`), or (iv) `delete` on `pods` (which removes the logs that `pods/log` exposes) or on the `pods/log` subresource as surveyed tooling checks it (`R-P3.2.4`), to a subject with no operational need to alter that configuration or erase that record.

Collected entries. Eight entries, split between the configuration side and the observability side. The second group is the reason the pattern is not summarised into `R-P3.1`. Deleting events and logs damages auditability without disclosing anything.

Entries	Detection (source)
#109, #139	manage and update ConfigMaps (Trivy, Kubescape)
#307	C-0258 — workload with ConfigMap access (Kubescape)
#319	ConfigMap holding sensitive values (Trivy)
#229	C-0037 — CoreDNS poisoning through ConfigMap write (Kubescape)
#154, #221	deletion of pod logs and of events (Trivy, Kubescape)
#206	excessive delete rights (Kubescape)

L3 — R-P3.2.1 ConfigMap write

A ConfigMap can contain configuration consumed by running components, making write access relevant to integrity rather than confidentiality. A particularly consequential example is the CoreDNS configuration stored in a ConfigMap in `kube-system`. A subject able to modify this configuration may alter cluster DNS resolution, potentially redirecting service names or other DNS queries to attacker-controlled destinations. This illustrates how apparently ordinary configuration write

permissions can affect the behaviour and security of other components without directly granting additional RBAC privileges.

The relevant RBAC operations include `update` and `patch`, whose impact depends on which ConfigMap is accessible and how its contents are consumed by the corresponding component. The pattern therefore captures configuration integrity risks rather than treating every ConfigMap write as an immediate privilege escalation.

✗ **Vulnerable manifest** (triggers R-P3.2.1)

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: coredns-editor
  namespace: kube-system
rules:
- apiGroups: [""]
  resources: ["configmaps"]
  verbs: ["update", "patch"]
```

L3 — R-P3.2.2 ConfigMap read

ConfigMaps are intended for non confidential configuration data and do not provide the Secret specific confidentiality guarantees associated with credential material. Nevertheless, in practice, ConfigMaps may contain connection strings, internal endpoints, or, in poorly configured deployments, credentials stored in plaintext. Read access can therefore expose information that a subject would not otherwise be expected to access.

The risk is particularly relevant for `get` and `list` permissions. A `list` permission over ConfigMaps can expose the contents of multiple configuration objects through a single API request, potentially revealing information across an entire namespace. This creates a separate exposure path from `R-P3.1`: unlike Secrets, ConfigMaps are not intended to represent sensitive credential material, but their contents may nevertheless contain data with comparable security implications.

✗ **Vulnerable manifest** (triggers R-P3.2.2)

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: configmap-reader
rules:
- apiGroups: [""]
  resources: ["configmaps"]
  verbs: ["get", "list"]
```

L3 — R-P3.2.3 Events deletion

Kubernetes Events provide a short lived operational record of cluster activity, including scheduling decisions, image-pull failures, container startup problems, and other significant state transitions. Deleting Events does not directly disclose information or grant additional privileges but its security impact lies instead in reducing the visibility available to operators and monitoring systems.

The `deletecollection` operation is particularly relevant because it can remove multiple Events from a namespace in a single request. This does not constitute a direct privilege escalation, since Events are transient by design and the API server audit log provides a separate record of API activity. However, audit logs do not

necessarily reproduce the operational information contained in Events. Deleting Events can therefore interfere with `kubectl get events` and with monitoring or alerting pipelines that consume the Events API, particularly in environments where audit logging is unavailable, incomplete, or not routinely monitored.

✗ VULNERABLE MANIFEST (triggers R-P3.2.3)

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: event-cleaner
rules:
- apiGroups: [""]
  resources: ["events"]
  verbs: ["delete", "deletecollection"]
```

L3 — R-P3.2.4 Pod log deletion

Container logs provide runtime evidence of application and system activity, and their removal can reduce the ability to investigate an ongoing or completed compromise. The manifest associated with this subpattern follows the detection rule as collected from the surveyed tooling (entry #154), which identifies `delete` on the `pods/log` subresource.

The Kubernetes API does not provide a `delete` operation for `pods/log`: the subresource is used to retrieve container logs. The actual destructive capability is instead associated with `delete` on the parent `pods` resource, which can remove the Pod and its locally associated runtime logs. Externally collected logs may, however, persist in a separate logging backend.

The discrepancy is retained as collected because it is relevant to the methodology of the taxonomy. In fact the subpattern documents what the surveyed detection rule checks, while the accompanying analysis distinguishes that rule from the capability actually exposed by the Kubernetes API. This prevents a tool-specific detection condition from being interpreted as evidence that the corresponding RBAC operation is valid or executable on the platform.

✗ VULNERABLE MANIFEST (triggers R-P3.2.4)

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: log-cleaner
rules:
- apiGroups: [""]
  resources: ["pods/log"]
  verbs: ["delete"]
```

Remediation.

- **ConfigMaps**: read and write access should be limited to subjects with an operational need, with write permissions restricted particularly tightly for ConfigMaps in system namespaces or those consumed by security- or infrastructure-critical components.
- **Events and logs**: read access should be granted according to operational requirements, while deletion capabilities should be restricted and monitored through audit logging. In particular, `deletecollection` on Events

and `delete` on Pods should not be broadly assigned, as they can reduce operational visibility or remove locally retained runtime evidence.

- **Controls**: preventive access controls should be combined with detection and audit mechanisms capable of identifying unauthorized configuration changes and attempts to remove security-relevant operational evidence.

5.5 Family R-F4 — Privilege Escalation Paths

This family has the most overlap with the other families in the taxonomy.

Effect. A limited foothold can be transformed into higher privilege through one or more additional authorization steps, potentially reaching cluster-admin. The rule does not directly grant the resulting privilege, instead, it grants a capability that can be used to obtain, assume, or confer that privilege. (ImpactScope: cluster).

Family definition

Family **R-F4** collects misconfigurations that open privilege escalation paths. These are Kubernetes authorization primitives through which a subject with a limited foothold can obtain or exercise additional privileges through one or more subsequent steps, potentially reaching full cluster control. Unlike **R-F2**, which captures permissions that directly over-authorize a subject, **R-F4** captures capabilities that enable a transition from an existing level of privilege to a higher one.

This family also has the greatest overlap with other families because the same permission can be both excessive in its own right and useful as an escalation primitive. For example, `R-P4.6.4` RBAC pivot overlaps with `R-P4.1`, while `R-P4.6.5` ServiceAccount-management pivot overlaps with `R-P1.3`. These overlaps are retained and documented as cross-boundary escalation vectors rather than treated as mutually exclusive classifications.

Threat narrative

R-F4 is the family through which an attacker can escalate everything. A typical real attack starts with an initial foothold obtained through another family (for example a compromised pod that yields a `R-P1.3` ServiceAccount), whose token happens to carry some **R-F4** permission and is then chained step by step up to cluster-admin. Observed chains include `R-P4.5` (`pods/exec`) with `R-P4.1` (`impersonate`) to act as cluster-admin, `R-P4.4` (`pods/create`) with `R-P4.6` to mount the host filesystem and read control plane credentials and `R-P4.1` (`bind`) with a cluster-admin binding to self-escalate.

MITRE ATT&CK mapping. In MITRE ATT&CK for Containers, the family sits under the Privilege Escalation tactic (`TA0004`).

Technique	Why it applies
Account Manipulation: Additional Container Cluster Roles T1098.006	a subject grants itself a higher-privileged binding
Container Administration Command T1609	R-P4.5 lets the attacker run commands inside a container
Event Triggered Execution T1546	R-P4.3 plants a webhook the API server invokes on every matching request, exactly the trigger this technique covers
Escape to Host T1611	R-P4.5+R-P4.6 let the attacker read the host filesystem
Deploy Container T1610	R-P4.4 lets the attacker create workloads directly

General detection criterion

Detection criterion — family R-F4

There exists an RBAC rule granting an escalation primitive (escalation verbs, credential issuance, admission control manipulation, workload creation or execution, or substrate-level access) to a subject without a legitimate operational need.

Patterns of family R-F4

Pattern	Name	L3 variants	Entries
R-P4.1	RBAC escalation verbs	R-P4.1.1 Bind R-P4.1.2 Escalate R-P4.1.3 Impersonate R-P4.1.4 Combined escalation verbs	10
R-P4.2	Credential issuance	R-P4.2.1 CSR approval R-P4.2.2 ServiceAccount token minting R-P4.2.3 CSR signers	5
R-P4.3	Admission-webhook control	R-P4.3.1 Mutating webhook control R-P4.3.2 Validating webhook control	3
R-P4.4	Workload creation	R-P4.4.1 Pod creation R-P4.4.2 Pod mutation	4
R-P4.5	Pod execution control	R-P4.5.1 exec R-P4.5.2 attach R-P4.5.3 portforward	6
R-P4.6	Cluster substrate access	R-P4.6.1 Kubelet API proxy R-P4.6.2 Service proxy bypass R-P4.6.3 PersistentVolume hostPath bypass R-P4.6.4 RBAC pivot R-P4.6.5 SA-management pivot	12

5.5.1 R-P4.1 — RBAC escalation verbs

Pattern definition. The verbs `bind`, `escalate`, and `impersonate` are privilege-escalation primitives because, when granted over sufficiently privileged targets, they can allow a subject to obtain or exercise privileges that were not directly assigned to it. Their effect ie mechanically different as seen in the sub patterns below.

Detection criterion — R-P4.1

A rule grants either (i) the verb `bind` on roles (`R-P4.1.1`), (ii) the verb `escalate` on roles (`R-P4.1.2`), (iii) the verb `impersonate` on users, groups or ServiceAccounts (`R-P4.1.3`), or (iv) two or more of the three together (`R-P4.1.4`), to a subject whose function does not require altering its own privilege boundary.

Collected entries. Ten entries. Notably, none of the surveyed tools checks `bind` or `escalate` on its own. Every check looks at the three verbs together, which is why `R-P4.1.1` and `R-P4.1.2` are stated on analytical grounds while the entries all sit on the combined variant.

Entries	Detection (source)
#17, #18	restrict escalation verbs in Roles (Kyverno)
#30, #43	`bind`, `escalate` and `impersonate` together (Kubescape)
#75	CIS 5.1.8 — limit use of Bind, Impersonate and Escalate (CIS)
#103, #108	impersonation of users and groups (Kubescape)
#80	impersonation of privileged groups (Trivy)
#200, #210	Helm Tiller, the documented in-cluster case (Kyverno)

L3 — R-P4.1.1 Bind

The `bind` permission allows a subject to create or modify a RoleBinding or ClusterRoleBinding for a role that the subject could not otherwise exercise directly. When the target role carries higher privileges, this provides a potential path to privilege escalation. The entry is primarily analytical on its own in the surveyed corpus while the combined condition detected by the tools is represented separately by `R-P4.1.4`.

✗ **Vulnerable manifest** (triggers R-P4.1.1)

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: binder
rules:
  - apiGroups: ["rbac.authorization.k8s.io"]
    resources: ["clusterroles"]
    verbs: ["bind"]
```

L3 — R-P4.1.2 Escalate

The `escalate` permission allows a subject to create or modify a Role or ClusterRole containing permissions that exceed the subject's existing authorization. It therefore removes the API server's normal anti-escalation restriction and can be used to construct a more privileged role. As with `bind`, this entry is primarily analytical in the surveyed corpus.

✗ **Vulnerable manifest** (triggers R-P4.1.2)

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: escalate-roles
rules:
  - apiGroups: ["rbac.authorization.k8s.io"]
    resources: ["clusterroles"]
    verbs: ["escalate"]
```

L3 — R-P4.1.3 Impersonate

The `impersonate` permission allows a subject to make an API request as another user, group, or ServiceAccount without possessing that identity's credentials. Unlike `bind` and `escalate`, it does not require modification of RBAC objects as the subject

submits the request with the appropriate impersonation headers and the API server evaluates the request under the impersonated identity.

The impact therefore depends directly on which identities can be impersonated. For example, permission to impersonate members of the `system` group can provide unrestricted administrative access to the API server. This case is tracked separately in the corpus as (entry #80). Impersonated requests are also represented in audit records, which preserves evidence of both the original and impersonated identities when audit logging is enabled.

✗ VULNERABLE MANIFEST (triggers R-P4.1.3)

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: impersonator
rules:
  - apiGroups: [""]
    resources: ["users", "groups", "serviceaccounts"]
    verbs: ["impersonate"]
```

L3 — R-P4.1.4 Combined escalation verbs

This is the variant actually detected by the surveyed tooling. None of the corresponding checks flags `bind` or `escalate` in isolation, nevertheless the corpus entries (entries #17, #18, #30, #43, #75) identify rules containing the three escalation verbs together.

The combined condition is therefore retained as a separate subpattern because it reflects the detection logic observed in the corpus. It should not, however, be interpreted as the minimum condition required for privilege escalation: `bind`, `escalate`, or `impersonate` can each provide an escalation path when applied to sufficiently privileged targets.

✗ VULNERABLE MANIFEST (triggers R-P4.1.4)

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: escalator
rules:
  - apiGroups: ["rbac.authorization.k8s.io"]
    resources: ["clusterroles", "clusterrolebindings"]
    verbs: ["bind", "escalate"]
  - apiGroups: [""]
    resources: ["users", "groups", "serviceaccounts"]
    verbs: ["impersonate"]
```

Remediation. Application workloads generally have no operational need for `bind`, `escalate`, or `impersonate`, so these permissions should be denied by default and granted only to tightly controlled administrative identities or automation with a documented operational need. Where they are required, the corresponding roles should be scoped as narrowly as possible, avoiding wildcard resources and API groups and restricting the set of roles or identities that can be bound or impersonated. In particular, impersonation permissions should be limited to explicitly required target identities and monitored through API audit logging.

The Kyverno [17] policy below provides a preventive admission control for these escalation verbs. Admission controls should complement, rather than replace, RBAC

review: every Role and ClusterRole containing `bind`, `escalate`, or `impersonate` should be reviewed periodically to verify that the permission remains necessary and that its scope has not expanded beyond the intended administrative boundary.

Enforcement policy

```yaml
apiVersion: kyverno.io/v1
kind: ClusterPolicy
metadata:
  name: restrict-escalation-verbs
spec:
  validationFailureAction: Enforce
  rules:
    - name: no-escalation-verbs
      match:
        any:
          - resources:
              kinds: ["Role", "ClusterRole"]
      validate:
        message: "Escalation verbs not allowed."
        foreach:
          - list: "request.object.rules"
            deny:
              conditions:
                any:
                  - key: "{{ element.verbs[] }}"
                    operator: AnyIn
                    value: ["bind", "escalate", "impersonate", "*"]
```

5.5.2 R-P4.2 — Credential issuance

Pattern definition. `R-P4.2` covers RBAC rules that grant control over the API's credential-issuing objects: `CertificateSigningRequests` and ServiceAccount tokens. The escalation lives in the issued credential itself rather than in the RBAC graph, so inspecting roles and bindings alone does not reveal it. None of its three L3 variants touches an administrative verb or references a privileged role directly, yet each produces a credential indistinguishable from a legitimate one.

Detection criterion — R-P4.2

A subject holds either (i) `create` on `certificatesigningrequests` together with `update` on `certificatesigningrequests/approval` and `approve` on the relevant `signers` entry (`R-P4.2.1`), (ii) `create` on `serviceaccounts/token` for a ServiceAccount bound to a more privileged role (`R-P4.2.2`), or (iii) `sign` on a named `signers` entry (`R-P4.2.3`), while the resulting certificate or token is never recorded as a change to an RBAC binding.

Collected entries. Five entries, evenly split between the two mechanisms, all with ImpactScope = Cluster.

Entries	Detection (source)
#124, #181	creation of ServiceAccount tokens (Kubescape)
#218, #233	approval of CertificateSigningRequests (Kubescape)
#12	ClusterRole covering CSRs (Kyverno)

L3 — R-P4.2.1 CSR approval

Approving a `CertificateSigningRequest` requires the verb `update` on `certificatesigningrequests/approval` together with the verb `approve` on `signers`, scoped by resource name to the target signer, for example `kubernetes.io/kube-apiserver-client`. A subject holding both becomes security-sensitive when it can also create a CSR (verb `create` on `certificatesigningrequests`) requesting an identity of its choice. Once approved, a request against a built-in client signer is signed automatically by the controller manager, and the resulting certificate can then be used to authenticate as that identity.

The escalation needs no privileged RoleBinding because the RBAC permissions above grant access to the issuance mechanism itself. The actual impact depends on the privileges of the certified identity, so holding these permissions alone does not automatically grant cluster-admin.

✗ **Vulnerable manifest** (triggers R-P4.2.1)

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: csr-approver
rules:
  - apiGroups: ["certificates.k8s.io"]
    resources: ["certificatesigningrequests"]
    verbs: ["create"]
  - apiGroups: ["certificates.k8s.io"]
    resources: ["certificatesigningrequests/approval"]
    verbs: ["update"]
  - apiGroups: ["certificates.k8s.io"]
    resources: ["signers"]
    verbs: ["approve"]
```

L3 — R-P4.2.2 ServiceAccount token minting

The verb `create` on `serviceaccounts/token` exposes the TokenRequest API, letting a subject obtain a short-lived token for a targeted ServiceAccount without holding its existing credentials and without touching the ServiceAccount or any RBAC binding.

The impact depends on how broadly the permission reaches. If it extends to ServiceAccounts bound to more privileged roles, for example those used by control-plane or infrastructure components, it becomes a path to substantially greater privilege, scaled by whatever those accounts can do. Each token is short-lived, but the subject can request a new one for as long as it retains the permission. The misconfiguration is therefore the permission itself reaching too broad a set of ServiceAccounts, while the token is only the credential that carries it out.

✗ **Vulnerable manifest** (triggers R-P4.2.2)

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: token-minter
rules:
  - apiGroups: [""]
    resources: ["serviceaccounts/token"]
    verbs: ["create"]
```

L3 — R-P4.2.3 CSR signers

The verb `sign` on the `signers` resource of `certificates.k8s.io` lets a subject sign a CSR for the specified signer, optionally scoped by `resourceNames` to a particular `signerName`.

The resulting certificate can represent an identity the subject could not otherwise obtain, but the impact depends on whether the API server trusts that signer's certificate authority for client authentication and on the privileges of the signed identity. This subpattern is therefore retained as an analytical RBAC pattern rather than a directly observed one. No surveyed tool checks for this permission.

✗ Vulnerable manifest (triggers R-P4.2.3)

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: csr-signer
rules:
  - apiGroups: ["certificates.k8s.io"]
    resources: ["signers"]
    verbs: ["sign"]
```

Remediation. Certificate approval, certificate signing, and ServiceAccount token issuance should be treated as administrative capabilities rather than general workload permissions. The corresponding RBAC permissions should be restricted to subjects with a documented operational requirement.

- **Certificate side**: permissions related to `certificatesigningrequests/approval` and sensitive `signers` operations should be excluded from workload roles wherever they are not required. Where custom signers are used, `approve` and `sign` permissions should be scoped as narrowly as possible, using `resourceNames` to constrain the permitted signer where supported. The Kyverno policy of entry #12, shown below, blocks the corresponding grant at admission.
- **Token side**: `create` on `serviceaccounts/token` should be restricted to the specific ServiceAccounts for which token issuance is operationally required. Where possible, the permission should be expressed through a namespaced Role rather than a ClusterRole to reduce the set of ServiceAccounts that can be targeted.
- **Audit trail**: certificate approval, certificate signing, and ServiceAccount TokenRequest operations should be monitored through API audit logging. Where request-body logging is enabled by the audit policy, the recorded request can provide additional information about the requested identity or target ServiceAccount, while metadata-only audit policies provide more limited visibility.

Enforcement policy

```yaml
# Kyverno (entry #12): deny a ClusterRole covering the CSR approval flow.
apiVersion: kyverno.io/v1
kind: ClusterPolicy
metadata:
  name: restrict-csr-approval
spec:
  validationFailureAction: Enforce
  rules:
```

```
- name: no-csr-approval-grant
  match:
    any:
      - resources:
          kinds: ["ClusterRole"]
  validate:
    message: "Approving CertificateSigningRequests is an operator capability
        ."
    foreach:
      - list: "request.object.rules"
        deny:
          conditions:
            all:
              - key: "{{ element.resources[] }}"
                operator: AnyIn
                value: ["certificatesigningrequests/approval"]
              - key: "{{ element.verbs[] }}"
                operator: AnyIn
                value: ["approve", "update"]
```

5.5.3 R-P4.3 — Admission-webhook control

Pattern definition. `R-P4.3` covers RBAC rules that grant control over the two webhook-configuration resources used to register admission webhooks: `MutatingWebhookConfiguration` and `ValidatingWebhookConfiguration`. The misconfiguration is an ordinary RBAC rule whose target is a sensitive resource such as an admission-control resource rather than an RBAC or application object.

Admission webhooks sit in the API request path for the resources and operations they match, so control over their configuration can affect how the API server processes later requests, without requiring direct RBAC permissions on the workloads those requests touch. The pattern therefore represents an indirect privilege-escalation path, where RBAC grants control over the admission layer, and that layer in turn shapes the security properties of resources other subjects create or modify.

Detection criterion — R-P4.3

A rule grants `create`, `update`, `patch` or `delete` either (i) on `mutatingwebhookconfigurations` (`R-P4.3.1`) or (ii) on `validatingwebhookconfigurations` (`R-P4.3.2`), to a subject other than the platform team or the controller that owns the configuration.

Collected entries. Three entries, all with ImpactScope = Cluster. None of them separates the mutating case from the validating one. The tools check control over webhook configurations as a whole, so the split into two L3 comes from the mechanism rather than from the data.

Entries	Detection (source)
#37	ClusterRole covering mutating and validating webhooks (Kyverno)
#197	manage webhook configurations (Trivy)
#222	modify admission webhooks (Kubescape)

L3 — R-P4.3.1 Mutating webhook control

The verbs `create`, `update`, or `patch` on `mutatingwebhookconfigurations` let a subject register or modify a webhook and place it in the admission path, where it

can modify an API object before it is persisted. A configuration matching broad resources and operations can put newly created Pods, for example, through code the subject controls, capable of altering properties such as `securityContext`, volumes, or `serviceAccountName`, subject to the webhook's own behaviour and to any other admission controls the cluster enforces.

A sufficiently broad configuration therefore gives an indirect route to workloads the subject could not modify directly through RBAC. The security boundary here is the admission mechanism, not the workload object itself. The `failurePolicy` field also affects the behaviour of the webhook when its endpoint cannot be reached. Setting `failurePolicy: Ignore` causes matching requests to continue when the webhook call fails, which can reduce the security effect of the webhook while leaving the configuration object present. This is particularly relevant when analysing modifications intended to weaken an existing admission configuration rather than replace it entirely.

✗ **Vulnerable manifest** (triggers R-P4.3.1)

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: webhook-controller
rules:
  - apiGroups: ["admissionregistration.k8s.io"]
    resources: ["mutatingwebhookconfigurations"]
    verbs: ["create", "update", "patch"]
```

L3 — R-P4.3.2 Validating webhook control

The verbs `update`, `patch`, or `delete` on `validatingwebhookconfigurations` let a subject weaken or remove checks that would otherwise reject dangerous configurations, including the Kyverno and Gatekeeper policies used elsewhere in the taxonomy to mitigate patterns such as `R-P1.1`, `R-P1.2`, and `R-P4.2`, since these are ultimately enforced through admission webhooks.

Deleting the configuration is the most direct route to stop the webhook from being invoked at all while modifying it is less visible because narrowing the `rules` field can exclude resources or operations from validation, and setting `failurePolicy: Ignore` lets requests proceed when the webhook endpoint fails, while the configuration object itself remains present in the cluster inventory. The resulting capability is therefore distinct from gaining additional permissions directly through RBAC. It instead removes or weakens an admission-layer control that constrains how other RBAC-authorised actions are performed.

✗ **Vulnerable manifest** (triggers R-P4.3.2)

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: webhook-gatekeeper
rules:
  - apiGroups: ["admissionregistration.k8s.io"]
    resources: ["validatingwebhookconfigurations"]
    verbs: ["create", "update", "patch"]
```

Remediation. Webhook configurations are cluster-wide control points and should normally be managed only by the platform team or by the controllers responsible

for the corresponding admission infrastructure.

- **RBAC level**: application roles should not receive `create`, `update`, `patch`, or `delete` permissions on either webhook-configuration resource. Where administrative access is required, the permissions should be restricted to dedicated platform identities and granted only for the required scope.
- **Admission protection**: admission policies can provide an additional layer of protection for webhook configuration objects, for example by restricting who can create or modify them. However, this control should not be the sole defence when the admission infrastructure itself is within the attacker's potential control path. The Kyverno policy of entry #37, shown below, therefore operates one level earlier by preventing the ClusterRole that grants the corresponding RBAC permission.
- **Detection**: changes to webhook configurations should be audited and monitored. Legitimate modifications are normally limited to controlled deployment or platform-management workflows, making unexpected `create`, `update`, `patch`, or `delete` operations suitable candidates for alerting and investigation.

Enforcement policy

```
# Kyverno (entry #37): deny a ClusterRole covering webhook configurations.
apiVersion: kyverno.io/v1
kind: ClusterPolicy
metadata:
  name: restrict-webhook-control
spec:
  validationFailureAction: Enforce
  rules:
    - name: no-webhook-configuration-grant
      match:
        any:
          - resources:
              kinds: ["ClusterRole"]
      validate:
        message: "Managing webhook configurations is reserved for the platform
            team."
        foreach:
          - list: "request.object.rules"
            deny:
              conditions:
                any:
                  - key: "{{ element.resources[] }}"
                    operator: AnyIn
                    value: ["mutatingwebhookconfigurations", "
                        validatingwebhookconfigurations"]
```

5.5.4 R-P4.4 — Workload creation

Pattern definition. Being able to create or modify Pods is an indirect form of privilege escalation because a Pod provides both an execution environment and an identity within the cluster. Creating a Pod (`R-P4.4.1`) lets a subject set properties such as its ServiceAccount, security context, network configuration, or mounted volumes, and Kubernetes itself recognises that permission to create workloads implicitly grants access to the privileges of the ServiceAccounts those workloads can run as [55]. Modifying an existing Pod (`R-P4.4.2`) is more constrained, since most of its specification becomes immutable after creation, but the fields that remain

mutable, or the `ephemeralcontainers` subresource where granted, can still change what code runs and what the Pod is reachable from.

The escalation therefore does not come from the verb itself, but from the capabilities that a workload created or modified through that permission can exercise.

Detection criterion — R-P4.4

A rule grants either (i) `create` on `pods` (or on a controller resource such as Deployments, Jobs or CronJobs that produces Pods, `R-P4.4.1`), or (ii) `update`/`patch` on `pods` (`R-P4.4.2`), to a subject other than the delivery pipeline or the controllers that manage those workloads.

Collected entries. Four entries. Three of them cover Pod creation and only one covers mutation, which matches the general pattern in the data. Creation verbs are checked far more often than update and patch.

Entries	Detection (source)
#219, #249	`create` on Pods (Kubescape)
#246	CIS 5.1.4 — minimise access to create pods (CIS)
#173	updating an existing Pod with a malicious spec (Trivy)

L3 — R-P4.4.1 Pod creation

A Pod provides both an execution environment and a Kubernetes identity, so `create` on `pods` gives the subject control over important properties of the workload it creates. In particular, the subject can select a ServiceAccount already associated with additional RBAC privileges and can request security-sensitive settings such as `privileged`, `hostNetwork`, or `hostPID`. It can also request volumes such as `hostPath`, subject to the admission and authorization controls enforced by the cluster.

The resulting impact depends on the namespace and on the admission controls applied to the Pod. Pod Security admission can reject host-level settings such as privileged containers, host namespaces, and hostPath volumes under appropriate enforcement levels but tt does not determine which ServiceAccount a workload is allowed to use. Consequently, the ability to create a Pod may still provide an escalation path even when the stronger host-level configurations are blocked.

The same capability can be obtained indirectly through `create` on controller resources such as Deployments, Jobs, and CronJobs, since these resources can produce Pods.

✗ **Vulnerable manifest** (triggers R-P4.4.1)

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: pod-creator
rules:
  - apiGroups: [""]
    resources: ["pods"]
    verbs: ["create"]
```

L3 — R-P4.4.2 Pod mutation

Most of a Pod's specification, including its ServiceAccount, volumes, and most security-related fields, becomes immutable once the object exists, so `update` and `patch` give a more constrained capability than `create`. What remains mutable can still matter: replacing the container image turns a Pod that already has a privileged ServiceAccount or sensitive mounted resources into one running code the subject chooses, while changing labels or annotations can alter which Services or NetworkPolicies select it. Where separately authorised, the `pods/ephemeralcontainers` subresource adds another execution path, letting an ephemeral container be injected into a running Pod.

The security impact therefore depends on both the mutable fields the permission exposes and the existing privileges of the targeted Pod.

✗ VULNERABLE MANIFEST (triggers R-P4.4.2)

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: pod-mutator
rules:
  - apiGroups: [""]
    resources: ["pods"]
    verbs: ["update", "patch"]
```

Remediation. `create` on `pods` and `update` or `patch` on Pods should normally be limited to workload controllers and controlled delivery identities rather than being assigned broadly to human or application identities. Entry #246 (CIS 5.1.4) expresses the same least-privilege principle and pairs it with Pod Security enforcement at the `baseline` or `restricted` level.

5.5.5 R-P4.5 — Pod execution control

Pattern definition. These misconfigurations provide interactive access to running Pods rather than to the API objects that describe them. The verbs `exec` (`R-P4.5.1`) and `attach` (`R-P4.5.2`) reach a container's own process, so a subject exercises whatever identity and mounted resources that container already holds. `portforward` (`R-P4.5.3`) instead opens a tunnel to a Pod's port through the API server, bypassing the normal Service path and the NetworkPolicy controls that govern it.

None of these permissions modifies RBAC objects but their potential escalation comes from entering an already privileged execution context or reaching an internal endpoint through an API-mediated connection rather than the network path a subject would otherwise be confined to.

DETECTION CRITERION — R-P4.5

A rule grants `create` either (i) on `pods/exec` (`R-P4.5.1`), (ii) on `pods/attach` (`R-P4.5.2`), or (iii) on `pods/portforward` (`R-P4.5.3`) to a ServiceAccount rather than to a named human operator.

Collected entries. Six entries and the only pattern of the family with a runtime detection alongside the static checks, which fits a set of verbs that leave a trace

while they are being used.

Entries	Detection (source)
#111, #162	shell or `exec` into a container (Trivy, Kubescape)
#232	attach to a Pod and read its streams (Trivy)
#195, #223	port-forward privileges (Kubescape)
#250	terminal shell in container, runtime detection (Falco)

L3 — R-P4.5.1 exec

The `create` verb on the `pods/exec` subresource starts a process inside a running container. The process executes within the same container context and can access resources available to that container, including mounted volumes and, where present, ServiceAccount credentials.

The permission therefore reaches as far as the most privileged Pod that the subject is authorised to target. This makes it particularly relevant when Pods are running under ServiceAccounts with broader privileges than the calling identity, creating a direct connection with `R-P1.3.2`. No new Kubernetes object or RBAC binding is required, so the escalation may not be visible through a comparison of cluster manifests.

✗ **Vulnerable manifest** (triggers R-P4.5.1)

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: pod-exec
rules:
  - apiGroups: [""]
    resources: ["pods/exec"]
    verbs: ["create"]
```

L3 — R-P4.5.2 attach

Where `exec` starts a new process, `create` on `pods/attach` connects to the streams of an existing process. It is therefore more constrained than `exec`, since the subject cannot select an arbitrary command to execute. Its impact nevertheless depends on the process already running in the container. If that process provides an interactive shell or another controllable interface, attaching to it may provide an interactive session under the container's existing identity.

The operation can also expose data written to the attached streams that is not necessarily represented in the container's ordinary log output.

✗ **Vulnerable manifest** (triggers R-P4.5.2)

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: pod-attach
rules:
  - apiGroups: [""]
    resources: ["pods/attach"]
    verbs: ["create"]
```

L3 — R-P4.5.3 portforward

The `create` verb on `pods/portforward` opens a tunnel from the client to a port on a Pod through the Kubernetes API. The connection does not use the normal Service routing path and therefore should not be treated as equivalent to ordinary pod-to-pod traffic.

This can expose internal application endpoints to a subject that otherwise could not reach them directly. For example, a database or administrative interface that is not externally exposed may still become reachable through the API server when the subject has permission to port-forward to the corresponding Pod. The precise interaction with NetworkPolicy depends on the networking implementation and on where the forwarded traffic enters the network path.

The subpattern therefore represents both an RBAC exposure and a cross-domain interaction with the network controls discussed in Chapter 7.

✗ **Vulnerable manifest** (triggers R-P4.5.3)

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: pod-portforward
rules:
  - apiGroups: [""]
    resources: ["pods/portforward"]
    verbs: ["create"]
```

Remediation. Interactive access to a running Pod is hard to prevent outright, so remediation focuses on restricting who can obtain it and on detecting when it is used.

- **Access**: `exec`, `attach`, and `portforward` should be treated as administrative or debugging capabilities rather than ordinary workload permissions. They should generally be granted to controlled human or support identities and kept out of general-purpose ServiceAccounts. Where access to a specific Pod is sufficient, the RBAC rule can be narrowed with `resourceNames`, which Kubernetes supports for these subresources.

- **Detection**: use API audit logging to monitor access to `pods/exec`, `pods/attach`, and `pods/portforward`. Runtime detection can complement this by identifying shells or other interactive processes created inside containers. The combination provides visibility into both the authorization event and its runtime consequence.

5.5.6 R-P4.6 — Cluster substrate access

Pattern definition. This pattern gathers RBAC permissions that provide access to cluster substrates or control points beneath ordinary workload objects. The covered mechanisms include the kubelet API exposed through `nodes/proxy`, API-server proxying to Services, PersistentVolumes, RBAC bindings, and ServiceAccount management.

These permissions differ in their escalation mechanism. Some provide direct access to infrastructure capabilities, such as the kubelet API. Others provide an

intermediate capability that requires an additional authorization step, such as creating a binding for an existing role. The common property is that the permission can extend a subject's effective reach beyond the workload object on which the permission is initially granted.

Detection criterion — R-P4.6

A rule grants either (i) `get` or `create` on `nodes/proxy` (`R-P4.6.1`), (ii) on `services/proxy` (`R-P4.6.2`), (iii) `create` on `persistentvolumes` (`R-P4.6.3`), (iv) `create` on `rolebindings` or `clusterrolebindings` (`R-P4.6.4`), or (v) `create`/`update` on `serviceaccounts` (`R-P4.6.5`), each reaching a layer below the object model that ordinary RBAC rules are meant to govern.

Collected entries. Twelve entries, the second most populated pattern of the family. They cluster around two mechanisms, the node proxy and the ability to write new bindings and this is where the overlap with `R-P4.1` and `R-P1.3` shows up in the data as well as in the definitions.

Entries	Detection (source)
#8, #9	ClusterRole with `nodes/proxy` (Kyverno)
#141, #194, #240	access to the node proxy sub-resource (Trivy, Kubescape)
#10, #55, #96	creating bindings that attach a privileged ClusterRole (Trivy)
#25	manage RBAC resources (Trivy)
#220	`create` on PersistentVolumes (Kubescape)
#31, #57	ServiceAccount management (Kyverno)

L3 — R-P4.6.1 Kubelet API proxy

The `nodes/proxy` subresource forwards requests through the API server to the HTTP API exposed by the kubelet on the selected node. Kubernetes documentation explicitly identifies this subresource as sensitive: access to `nodes/proxy` can provide access to kubelet APIs capable of retrieving container information and executing or attaching to processes on Pods running on the node. In particular, `get` is not equivalent to a conventional read-only permission for this resource because some kubelet operations are reachable through HTTP GET requests. [55]

A subject with this permission can therefore reach workloads on the node without holding the corresponding `pods/exec` or `pods/attach` permissions through the Kubernetes API. The effective impact depends on the kubelet's authentication and authorization configuration. A kubelet configured with an excessively permissive authorization mode can therefore amplify the impact of the `nodes/proxy` permission.

✗ **Vulnerable manifest** (triggers R-P4.6.1)

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: node-proxy
rules:
  - apiGroups: [""]
    resources: ["nodes/proxy"]
    verbs: ["get", "create"]
```

L3 — R-P4.6.2 Service proxy access

`get` or `create` on `services/proxy` allows a subject to use the API server as a proxy to a Service and its backend endpoints. Unlike direct access from the caller's network location, the request is mediated by the API server and can reach a ClusterIP service that is not otherwise exposed to the caller.

This can reach internal application endpoints that would otherwise be unreachable. Whether it bypasses NetworkPolicy depends on the CNI implementation, since the connection does not follow the ordinary client-to-Service path and may hit different enforcement points along the way.

No surveyed tool checks this permission, and the subpattern has no matching entry in the corpus. It is retained as an analytical pattern based on the architecture of the API-server proxy, not as an empirically observed detection rule.

✗ **Vulnerable manifest** (triggers R-P4.6.2)

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: service-proxy
rules:
  - apiGroups: [""]
    resources: ["services/proxy"]
    verbs: ["get", "create"]
```

L3 — R-P4.6.3 PersistentVolume hostPath bypass

`create` on `persistentvolumes`, a cluster-scoped resource, allows a subject to define a PersistentVolume backed by a `hostPath`, namely a directory on a node. Kubernetes explicitly identifies arbitrary PersistentVolume creation as a security risk because a `hostPath` volume can expose the underlying node filesystem to a Pod. [55]

The permission is not sufficient by itself to read the host filesystem and the reason is that the PersistentVolume must be claimed and mounted by a workload. The escalation path therefore requires additional capabilities, such as the ability to create or control a PersistentVolumeClaim and a Pod. Once mounted, however, the host filesystem may expose sensitive system credentials or other node-level resources. [58]

This route is particularly relevant when direct `hostPath` volumes in Pod specifications are blocked by admission controls. The Pod can instead refer to a PersistentVolumeClaim while the `hostPath` is defined in the PersistentVolume object. The overlap with **R-F3** is therefore primarily one of consequence rather than of authorization predicate.

✗ **Vulnerable manifest** (triggers R-P4.6.3)

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: pv-creator
rules:
  - apiGroups: [""]
    resources: ["persistentvolumes"]
    verbs: ["create"]
```

L3 — R-P4.6.4 RBAC pivot

The `create` verb on `rolebindings` and `clusterrolebindings` allows a subject to create new bindings that reference existing Roles or ClusterRoles. Kubernetes normally prevents a subject from using this capability to grant permissions it does not already possess. The exception is the `bind` permission described in `R-P4.1.1`, which allows this protection to be bypassed for the specified role.

Consequently, the creation permission is not by itself equivalent to privilege escalation. When combined with `bind`, it can provide a direct escalation path by attaching a more privileged role to an identity controlled by the subjectb. Without `bind`, the same capability may still be security-sensitive because it allows a subject to create additional bindings for permissions that it already possesses, potentially providing persistence or delegating those permissions to another identity.

The distinction is retained in the taxonomy because the RBAC permission and the escalation primitive are separate conditions: `create` on bindings provides the mechanism for writing the new binding, while `bind` determines whether a higher-privileged role can be attached.

✗ **Vulnerable manifest** (triggers R-P4.6.4)

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: binding-creator
rules:
  - apiGroups: ["rbac.authorization.k8s.io"]
    resources: ["rolebindings", "clusterrolebindings"]
    verbs: ["create"]
```

L3 — R-P4.6.5 ServiceAccount-management pivot

Permissions to create or modify ServiceAccounts can become part of an escalation chain, but they do not by themselves provide the privileges of another ServiceAccount. The security impact depends on how the ServiceAccount is subsequently used and on which additional permissions are available to the subject.

For example, a subject able to create ServiceAccounts and also able to request tokens through `serviceaccounts/token` may be able to create an identity and subsequently obtain a credential for it. Similarly, in an environment where other bindings or controllers associate privileges with ServiceAccounts created or modified by the subject, ServiceAccount management can become an intermediate escalation capability.

The subpattern should therefore be interpreted as an enabling condition rather than as an unconditional privilege escalation. Its overlap with `R-P1.3` is also conditional: the existence of a privileged ServiceAccount does not make arbitrary ServiceAccount creation equivalent to taking over that identity, since RBAC bindings are attached to the ServiceAccount identity and are not transferred to newly created accounts.

✗ **Vulnerable manifest** (triggers R-P4.6.5)

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: sa-manager
rules:
```

```
  - apiGroups: [""]
    resources: ["serviceaccounts"]
    verbs: ["create", "update"]
```

Remediation. Each member of this pattern requires a different control, but the common principle is to keep infrastructure-level permissions outside ordinary workload identities.

- **Node and service proxy** (`R-P4.6.1`, `R-P4.6.2`): keep `nodes/proxy` and `services/proxy` out of ordinary application roles and restrict them to narrowly scoped administrative identities. The kubelet should use an authorization mode that performs proper authorization checks rather than an unrestricted mode such as `AlwaysAllow`.
- **Storage** (`R-P4.6.3`): restrict `create` on `persistentvolumes` to trusted storage administrators and infrastructure controllers. Where workloads require persistent storage, prefer controlled provisioning through StorageClasses and PVCs rather than allowing arbitrary PV definitions.
- **Binding and ServiceAccount pivots** (`R-P4.6.4`, `R-P4.6.5`): restrict the ability to create or modify bindings and ServiceAccounts to identities with a documented operational need. In particular, review combinations involving `bind`, `serviceaccounts/token`, and privileged ServiceAccounts rather than treating each permission as an isolated condition.

The Kyverno policies of entries #8 and #9 (`R-P4.6.1`) and entries #31 and #57 (`R-P4.6.5`) enforce part of this at admission, by refusing a ClusterRole that grants `nodes/proxy` or that manages ServiceAccounts.

Enforcement policy

```
# Kyverno (entries #8, #9): deny a ClusterRole granting nodes/proxy.
# Kyverno (entries #31, #57): deny a role managing ServiceAccounts.
apiVersion: kyverno.io/v1
kind: ClusterPolicy
metadata:
  name: restrict-substrate-access
spec:
  validationFailureAction: Enforce
  rules:
    - name: no-node-proxy-grant
      match:
        any:
          - resources:
              kinds: ["ClusterRole"]
      validate:
        message: "Access to nodes/proxy is reserved for the control plane."
        foreach:
          - list: "request.object.rules"
            deny:
              conditions:
                any:
                  - key: "{{ element.resources[] }}"
                    operator: AnyIn
                    value: ["nodes/proxy"]
    - name: no-serviceaccount-management-grant
      match:
        any:
          - resources:
              kinds: ["Role", "ClusterRole"]
      validate:
        message: "Creating or updating ServiceAccounts is an operator capability
```

```
            ."
        foreach:
          - list: "request.object.rules"
            deny:
              conditions:
                all:
                  - key: "{{ element.resources[] }}"
                    operator: AnyIn
                    value: ["serviceaccounts"]
                  - key: "{{ element.verbs[] }}"
                    operator: AnyIn
                    value: ["create", "update"]
```

Final note. R-F4 represents the principal privilege-escalation surface identified in the RBAC taxonomy. Its members differ substantially in mechanism: some provide direct access to privileged execution contexts, some control admission or infrastructure layers, and others enable a second authorization step. The common property is that the RBAC permission can transform an existing foothold into access that exceeds the subject's original authorization.

5.6 Family R-F5 — Authorization Layer Preconditions

> ***R-F5*** *groups 29 entries in total. It is RBAC-adjacent by definition and it is included because this family sets the prerequisite for a functional system and without it RBAC would not hold.*

Effect. The authorization layer is weakened or incompletely configured, so the authorization boundaries assumed by the RBAC model may not be applied as intended (ImpactScope: cluster).

Family definition

The main aspect of this family is that unlike the previous ones which describe proper RBAC misconfigurations (rules, bindings, capabilities), **R-F5** represents control plane configuration (`kube-apiserver`, `kubelet`, `kube-controller-manager`) without which RBAC is ignored, bypassed, or not enforced.

These conditions are therefore not RBAC rules themselves. They are included in the taxonomy because a correctly configured Role or ClusterRole cannot provide its intended security guarantee if the component responsible for evaluating or applying the authorization boundary is misconfigured.

Threat narrative

Rather than a rule a subject can exploit directly, **R-F5** is a precondition. Get it wrong and the RBAC model itself stops being enforced, no matter how correctly every other rule and binding is written. The usual case is a cluster deployed badly with `AlwaysAllow` in `-authorization-mode`, a development shortcut that reaches production. It means that RBAC is there but it is bypassed, so any foothold escalates to cluster-admin on its own.

MITRE ATT&CK mapping. In MITRE ATT&CK for Containers, **R-F5** maps to the precondition behind many other techniques rather than to a technique of its own.

Technique	Why it applies
Valid Accounts T1078	anonymous requests are treated as valid accounts
Deploy Container T1610	a permissive API server lets any container be deployed

General detection criterion

Detection criterion — family R-F5

A control plane component is configured so that RBAC is not enforced for some class of requests (`authorization-mode` omits `RBAC` or includes `AlwaysAllow`, or the kubelet allows anonymous authentication).

Patterns of family R-F5

Pattern	Name	L3 variants	Entries
`R-P5.1`	RBAC authorizer disabled	`R-P5.1.1` apiserver authorization-mode without RBAC `R-P5.1.2` apiserver authorization-mode includes AlwaysAllow `R-P5.1.3` kubelet authorization-mode AlwaysAllow `R-P5.1.4` kubelet anonymous-auth=true `R-P5.1.5` Cloud-managed equivalent	22
`R-P5.2`	Node authorizer missing	—	4
`R-P5.3`	Controller-manager SA misconfig	—	3

5.6.1 R-P5.1 — RBAC authorizer disabled or bypassed

Pattern definition. `R-P5.1` covers configurations in which authorization for the Kubernetes API or kubelet API is disabled or made substantially less restrictive. The pattern is divided into four forms:

- the apiserver `authorization-mode` does not include `RBAC`;
- it includes `AlwaysAllow`;
- the kubelet mode includes `AlwaysAllow`;
- the kubelet has `anonymous-auth=true`.

The last condition is an authentication configuration rather than an authorization mode by itself. It is retained in this pattern because, when combined with permissive kubelet authorization or an authorization rule for the anonymous identity, it can turn an unauthenticated network request into an authorised kubelet operation.

Detection criterion — R-P5.1

`kube-apiserver` is started with an `authorization-mode` list that either (i) omits `RBAC` (`R-P5.1.1`) or (ii) includes `AlwaysAllow` (`R-P5.1.2`), or the kubelet is started either (iii) with `authorization-mode=AlwaysAllow` (`R-P5.1.3`) or (iv) with `anonymous-auth=true` (`R-P5.1.4`), or (v) a cloud provider's authorization feature that the deployment's own threat model requires is left disabled on a managed cluster, leaving native RBAC as the only layer where more than one was intended (`R-P5.1.5`).

Collected entries. Twenty-two entries across five L3 (`R-P5.1.1`–`R-P5.1.5`). The first four are the raw apiserver and kubelet flags above, and `R-P5.1.5` is their cloud-managed equivalent. It is the most populated pattern of the taxonomy and the reason why is that a flag on a static manifest is the easiest thing a scanner can check.

Entries	Detection (source)
#26, #81, #142, #178	`authorization-mode` includes RBAC (Kubescape)
#40, #130	the same argument read on the apiserver (Trivy)
#112	CIS 1.2.8 — `authorization-mode` includes RBAC (CIS)
#93, #126, #134	`authorization-mode` not set to `AlwaysAllow` (Kubescape)
#168	CIS 1.2.6 — same check on the apiserver (CIS)
#83, #148	kubelet mode set to `AlwaysAllow` (Kubescape, CIS 4.2.2)
#298, #331, #368	anonymous access to the kubelet (Kubescape)
#321, #375	`anonymous-auth` on the kubelet (Trivy, CIS 4.2.1)
#191	guideline: limiting access to the kubelets (OWASP)
#73, #95, #159	RBAC on managed platforms, Azure and cloud (Kubescape)

L3 — R-P5.1.1 apiserver authorization-mode without RBAC

The API server `authorization-mode` configuration omits `RBAC`. Consequently, authorization decisions are made by the other configured authorizers rather than by Kubernetes RBAC. If no alternative authorizer provides the intended least-privilege policy, the Role and ClusterRole objects stored in the cluster cannot provide their normal authorization guarantees. Kubernetes supports multiple authorization modes, including `Node`, `RBAC`, `Webhook`, and `AlwaysAllow`. [51]

✗ **Vulnerable manifest** (triggers R-P5.1.1)

```
apiVersion: v1
kind: Pod
metadata:
  name: kube-apiserver
  namespace: kube-system
spec:
  containers:
  - name: kube-apiserver
    command:
    - kube-apiserver
    - --authorization-mode=Node,Webhook   # <-- RBAC omitted from the mode list
        (R-P5.1.1)
```

L3 — R-P5.1.2 apiserver authorization-mode includes AlwaysAllow

The authorization configuration contains `AlwaysAllow`. This authorizer returns an allow decision for requests that reach it, so the authorization boundary provided by RBAC cannot provide an effective restriction for those requests. The effect is especially direct when `AlwaysAllow` is the only authorizer or precedes RBAC in the authorization chain. Kubernetes explicitly warns that `AlwaysAllow,RBAC` has the same effective behaviour as `AlwaysAllow`, because RBAC cannot express deny rules. [51]

✗ **Vulnerable manifest** (triggers R-P5.1.2)

```
apiVersion: v1
kind: Pod
metadata:
  name: kube-apiserver
  namespace: kube-system
spec:
  containers:
  - name: kube-apiserver
    command:
    - kube-apiserver
    - --authorization-mode=AlwaysAllow   # <-- RBAC bypassed (R-P5.1.2)
```

L3 — R-P5.1.3 kubelet authorization-mode AlwaysAllow

The kubelet is configured with `authorization-mode=AlwaysAllow`, causing authenticated requests to the kubelet API to be authorised without the normal SubjectAccessReview checks provided by `Webhook` authorization. Kubernetes documents `AlwaysAllow` and `Webhook` as the two kubelet authorization modes, with `Webhook` delegating authorization decisions to the API server. [52]

The impact depends on which network clients can reach the kubelet HTTPS endpoint and how kubelet authentication is configured. The condition becomes particularly severe when anonymous authentication is also enabled.

✗ **Vulnerable manifest** (triggers R-P5.1.3)

```
apiVersion: kubelet.config.k8s.io/v1beta1
kind: KubeletConfiguration
authorization:
  mode: AlwaysAllow            # <-- kubelet authorizes everything (R-P5.1.3)
```

L3 — R-P5.1.4 kubelet anonymous-auth=true

The kubelet accepts requests that carry no credential, attributing them to `system:anonymous` and to the group `system:unauthenticated`. On its own the flag decides authentication and not authorization, so its effect depends on what follows. With the kubelet in `Webhook` mode an anonymous request is still checked, whereas paired with `R-P5.1.3` it authorizes anyone able to reach the node on port 10250 and paired with a binding to the anonymous group (`R-P1.2.1`) it grants that group whatever the binding names.

✗ **Vulnerable manifest** (triggers R-P5.1.4)

```
apiVersion: kubelet.config.k8s.io/v1beta1
kind: KubeletConfiguration
authentication:
  anonymous:
    enabled: true                # <-- anonymous kubelet access (R-P5.1.4)
```

L3 — R-P5.1.5 Cloud-managed equivalent

On managed clusters the authorization layer may be abstracted behind the provider's own tooling. Azure RBAC for Kubernetes Authorization, for instance, is an opt-in feature that some deployments rely on as an additional layer over native RBAC, for example to keep authorization policy consistent with the provider's own identity system. Native RBAC alone is not itself a misconfiguration, since the finding only applies when the deployment's own security baseline calls for that extra layer and it is left disabled, silently falling back to native RBAC alone.

Detection criterion — R-P5.1.5

A managed cluster's provider-specific authorization feature, required by the deployment's own documented baseline, is not enabled, leaving only native RBAC active where more than one layer was intended.

Remediation.

- **API server**: configure the API server so that `RBAC` is included in the authorization configuration and remove `AlwaysAllow` unless there is a documented and isolated testing requirement.
- **Kubelet**: configure `authorization.mode=Webhook` and disable anonymous authentication with `authentication.anonymous.enabled=false` where anonymous access is not explicitly required. Kubernetes documents `Webhook` as the kubelet authorization mechanism that delegates authorization to the API server. [52]
- **Managed clusters**: verify the provider-specific authorization configuration separately from native Kubernetes RBAC. Enabling a provider-level authorization mechanism should not be treated as a replacement for verifying the native RBAC configuration unless the provider explicitly documents that interaction.

5.6.2 R-P5.2 — Node authorizer missing

Pattern definition. The `Node` authorizer is a specialised authorization mode for kubelet requests, confining a kubelet to what its own node needs [54]: it may read a Secret, a ConfigMap or a PersistentVolume only if a Pod scheduled on that node refers to it. When `Node` is absent from the `authorization-mode` list, kubelet requests are instead evaluated by the remaining authorizers, such as RBAC alone, and the roles held by the node group are not written per node, so one compromised kubelet reads what every other node holds as well. The misconfiguration grants no permission of its own but it removes the boundary that kept an existing permission local, which is what places it among the preconditions rather than among the escalation paths of **R-F4**.

Detection criterion — R-P5.2

`kube-apiserver`'s `authorization-mode` list omits `Node`, so kubelet requests are authorised by RBAC alone and the per-node boundary the `Node` authorizer would otherwise enforce is not applied.

Collected entries. Four entries, all reading the same apiserver argument. The agreement across three independent sources is what makes the pattern solid despite its size.

Entries	Detection (source)
#102, #132	`authorization-mode` includes `Node` (Kubescape)
#104	the same argument read on the apiserver (Trivy)
#169	CIS 1.2.7 — `authorization-mode` includes `Node` (CIS)

✗ **Vulnerable manifest** (triggers R-P5.2)

```
apiVersion: v1
kind: Pod
metadata:
  name: kube-apiserver
  namespace: kube-system
spec:
  containers:
  - name: kube-apiserver
    command:
    - kube-apiserver
    - --authorization-mode=RBAC   # <-- Node authorizer missing (R-P5.2)
```

Remediation. Add `Node` to `-authorization-mode` alongside `RBAC` and enable the `NodeRestriction` admission plugin. Together the two confine a kubelet's credential to the objects referenced by Pods scheduled on its own node, restoring the per-node boundary whose absence otherwise lets one compromised kubelet read what every other node holds as well. Kubernetes explicitly recommends the combination for limiting kubelet write access [54].

5.6.3 R-P5.3 — Controller-manager SA misconfig

Pattern definition. The controller manager runs dozens of independent control loops and with `-use-service-account-credentials=true` each of them authenticates as its own ServiceAccount in `kube-system`, bound to the small `system:controller:*` role that loop needs [57]. Setting the flag to `false` makes them all share the controller manager's own credential, which then has to cover the union of what every loop does. The partition that RBAC was applying inside the control plane disappears and a defect in any single loop operates with the aggregate permission set. Moreover the audit log attributes every controller action to one identity, so it no longer records which loop performed it. As in `R-P5.2`, nothing new is granted. What is lost is a separation the rest of the model assumes.

Detection criterion — R-P5.3

`kube-controller-manager` is started with `-use-service-account-credentials=false`, so every control loop authenticates with the controller manager's own credential instead of a per-loop ServiceAccount. The audit log can then no longer attribute an action to the loop that performed it.

Collected entries. Three entries, one per source, all on the same controller-manager flag.

Entries	Detection (source)
#188	`use-service-account-credentials` on the controller manager (Trivy)
#193	C-0146 — the same flag (Kubescape)
#227	CIS 1.3.3 — the same flag (CIS)

✗ Vulnerable manifest (triggers R-P5.3)

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: kube-controller-manager
  namespace: kube-system
spec:
  containers:
  - name: kube-controller-manager
    command:
    - kube-controller-manager
    - --use-service-account-credentials=false    # <-- R-P5.3
```

Remediation. Set `-use-service-account-credentials=true` so that each controller uses its dedicated ServiceAccount and the corresponding least-privilege permissions. Kubernetes provides dedicated `system:controller:*` roles for the built-in controllers when this model is used. [57]

The setting should be treated as a control-plane hardening requirement rather than as an RBAC rule. The configuration is established in the controller-manager deployment or control-plane manifest and should therefore be verified through configuration auditing or benchmark scanning.

Entries #188, #193, and #227 (CIS 1.3.3) all provide examples of this configuration check.

Chapter 6

NetworkPolicy Misconfigurations

This chapter presents the NetworkPolicy misconfigurations taxonomy. It follows the same bottom-up method introduced in Chapter 4 and already applied to RBAC in Chapter 5, and it groups entries by the same principle, the nature of the failure. Where RBAC governs what an identity may do on the control plane, NetworkPolicy governs what a pod may reach on the data plane. The taxonomy is articulated into six families:

N-F1 — **Missing Isolation** pods are reachable because no policy isolates them. Kubernetes is allow-all by default. If no NetworkPolicy selects a pod, all ingress and egress are permitted.

N-F2 — **Over-permissive Connectivity** a policy exists but allows far too much: an `ipBlock` of `0.0.0.0/0`, over-broad selectors, all ports, or unrestricted egress/DNS.

N-F3 — **Ineffective Policy** a policy exists but has no effect: a selector that matches nothing, additive (no-deny) semantics, or precedence mistakes.

N-F4 — **Sensitive Endpoint Exposure** a sensitive endpoint (cloud metadata, the kubelet, a database, a dashboard) is reachable through the net when it should be fenced off.

N-F5 — **Enforcement Layer Preconditions** the layer beneath the policies fails, whether because the CNI does not implement NetworkPolicy or because the pod leaves the pod network, and a correct policy has no effect either way.

N-F6 — **CNI-extension & Admin Policy** failures specific to CNI policy CRDs (Cilium, Calico) and cluster-wide `AdminNetworkPolicy`, which go beyond the standard `NetworkPolicy` object and the reference scheme. Being the only family with no entries in the survey, it is developed in Appendix A.

6.1 Corpus statistics

The NetworkPolicy corpus comprises **124** collected entries (detection rules, policy recipes and guideline pages, see Chapter 4). Manual review maps them onto **6 families** and **15 patterns**, 9 backed by a detector drawn from the survey and 6 stated on analytical grounds without one. These 124 entries are not a flat list of

distinct misconfigurations, because part of them consists of near duplicates, that is, different tool rules that flag the same defect, of complementary controls that fall outside the core NetworkPolicy axis, or of reference example manifests.

Of these 124 entries, **40** survive as distinct pattern instances once near duplicates are collapsed. This is the figure the per-family counts in Table 6.1 are built on. The near duplicates removed in this step, marked in the corpus as `DuplicateOf` the entry they repeat, number **24**. Not everything left out of the 40 is noise, however. **14** entries are borderline or complementary material, such as L7 rules, service mesh controls or Ingress objects, that remains outside the core NetworkPolicy axis but is kept as context. **41** entries, covering PodSecurity, RBAC, example manifests or deprecated resources, fall outside the scope of the taxonomy and are excluded. The remaining **5** entries stay unclassified for lack of a strong signal. Distinct, collapsed, borderline, excluded and unclassified entries together sum to the full **124** collected entries.

Bucket	**Entries**	**Meaning**
Total collected	124	raw survey entries (Chapter 4)
N-F1 Missing Isolation	12	mapped entries
N-F2 Over-permissive Connectivity	9	mapped entries
N-F3 Ineffective Policy	5	mapped entries
N-F4 Sensitive Endpoint Exposure	2	mapped entries
N-F5 Enforcement Layer Preconditions	12	mapped entries
N-F6 CNI-extension & Admin Policy	0	mapped entries
Residual (duplicates + borderline)	38	retained as context
Retained after review	**78**	40 family-mapped + 38 residual

Table 6.1: Data funnel for the NetworkPolicy domain: reduction from 124 collected entries to 78 retained entries.

N-F6 has no entries at all. Almost nothing in the surveyed tooling looks at CNI extensions or at `AdminNetworkPolicy`, so the three patterns of that family are stated on analytical grounds or documentation pages instead of being derived from a tool rule. The gap is worth reporting, because it says something about the tools rather than about clusters, though two of the three patterns are not left unsupported: `N-P6.1` and `N-P6.2` are each anchored to a published CVE, which is what justifies keeping the family in the taxonomy rather than dropping it for lack of evidence. Since the family rests on analysis and deals with objects outside the standard `NetworkPolicy` API, it is developed in Appendix A, and the rest of this chapter covers **N-F1** to **N-F5**.

6.2 Family N-F1 — Missing Isolation

12 distinct entries are assigned to this family across two patterns (10 for `N-P1.1`, 2 for `N-P1.2`).

Effect. Unisolated pods are reachable across the namespace, giving an attacker lateral movement once inside (ImpactScope: namespace).

Family definition

Family N-F1 collects misconfigurations in which a pod remains reachable because no effective isolation policy is applied to it. In Kubernetes, pods are non-isolated by default in the absence of applicable `NetworkPolicy` rules so without an ingress policy selecting a pod, it accepts whatever ingress traffic and without an egress policy selecting a pod, it can initiate egress traffic. Isolation must therefore be established explicitly, typically by deploying a namespace-level default-deny baseline.

N-F1 represents the failure to establish such an isolation baseline. The defect is therefore the absence of a required control rather than an incorrect rule. This distinguishes N-F1 from N-F2, where a policy is present but permits excessive traffic, and from N-F3, where a policy is present but does not effectively constrain the intended traffic.

Threat narrative

A cluster without effective network isolation provides a broad communication surface between workloads. In the absence of applicable `NetworkPolicy` rules, a compromised pod may be able to communicate with other pods and Services that would otherwise be intentionally isolated. An attacker who compromises a single application pod can therefore gain network-level reachability to additional workloads, including internal APIs and databases, depending on the cluster topology and exposed services.

Missing isolation is rarely the complete attack path by itself. Rather, it can amplify the impact of an initial compromise by enabling lateral movement to workloads that should have been unreachable. This condition is particularly relevant to the cross-domain pattern `X1`, which composes network-level reachability with an overprivileged identity (Chapter 7)

MITRE ATT&CK mapping. In MITRE ATT&CK for Containers, the capability unlocked is the Lateral Movement tactic, supported by the following techniques.

Technique	Why it applies
Network Service Discovery `T1046`	a flat network lets an attacker enumerate exposed services
Container and Resource Discovery `T1613`	and then reach whatever it discovers

General detection criterion

Detection criterion — family N-F1

There exists a pod for which no `NetworkPolicy` produces a default-deny effect in at least one direction (that is, the pod accepts ingress from, or sends egress to, arbitrary peers cluster-wide).

Patterns of family N-F1

Pattern	Name	L3 variants	Entries
N-P1.1	Missing default-deny baseline	N-P1.1.1 No baseline at all N-P1.1.2 Ingress or Egress left open N-P1.1.3 Partial coverage	10
N-P1.2	Missing baseline on new namespaces	—	2

6.2.1 N-P1.1 — Missing default-deny baseline

Pattern definition. The namespace lacks an effective default-deny baseline, so pods that are not otherwise selected by restrictive `NetworkPolicy` rules may remain non-isolated. This represents the dominant form of missing isolation and is the most populated NetworkPolicy pattern in the dataset.

Detection criterion — N-P1.1

For a namespace ns there is no `NetworkPolicy` with an empty `podSelector` (`{}`) whose `policyTypes` cover the direction $d \in \{\mathsf{Ingress}, \mathsf{Egress}\}$ with no allow-rule, or at least one pod in ns is matched by no policy in direction d. In either case d defaults to allow-all.

Collected entries. Ten entries back this pattern, grouped by detection:

Entries	Detection (source)
#2, #3, #14, #18, #36, #43, #64	namespace without a NetworkPolicy (Kubescape [3])
#9, #21	guideline: use NetworkPolicies (OWASP [45])
#84	non-isolated pod — partial coverage N-P1.1.3 (kube-linter [49])

L3 — N-P1.1.1 No baseline at all

The namespace contains no `NetworkPolicy` objects, or no policy selects the affected pod. As a result, the pod is not isolated for either ingress or egress: it may receive traffic from any source and initiate traffic to any destination that is otherwise reachable at the network layer. This is the plainest and most severe form of missing isolation and, with 7 of the pattern's 10 entries, the dominant instance in the dataset.

✗ **Vulnerable manifest** (triggers N-P1.1.1)

```
# No NetworkPolicy selects this pod -> it accepts ALL ingress AND egress
# from any peer. The misconfiguration is the plain absence of a policy.
apiVersion: v1
kind: Pod
metadata:
  name: web
  namespace: prod      # <-- the namespace has no NetworkPolicy at all (N-P1
      .1.1)
spec:
  containers:
```

```
    - name: web
      image: nginx:1.27
```

L3 — N-P1.1.2 Ingress or Egress left open

A default-deny baseline covers one direction but leaves the other unaddressed, so that direction defaults to allow-all. In practice ingress is almost always the one addressed first, since it is the direction that closes outside reachability, which leaves egress as the direction most often forgotten (a very common partial configuration): pods cannot be reached from outside, but a compromised pod can still originate connections to any destination, which is exactly what enables data exfiltration and command-and-control. The reverse ordering, an egress-only baseline that leaves ingress unaddressed, is the same failure with the two directions swapped.

✗ **Vulnerable manifest** (triggers N-P1.1.2)

```
# Ingress is locked down, but Egress is NOT in policyTypes:
# the pod can still open outbound connections anywhere (exfiltration).
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: deny-ingress-only
  namespace: prod
spec:
  podSelector: {}
  policyTypes: ["Ingress"]   # <-- Egress omitted: egress stays allow-all (N-P1
      .1.2)
```

L3 — N-P1.1.3 Partial coverage

Policies are present, but their `podSelector`s do not match every pod in the namespace. A pod becomes isolated for a given traffic direction only when it is selected by at least one policy governing that direction and consequently, a pod that is not selected by any applicable policy remains non-isolated and may send or receive traffic without NetworkPolicy-based restrictions.

The gap typically appears after the baseline has been established. For example, a baseline that selects pods labelled `app:backend` does not select a newly deployed workload whose pods are labelled `tier:backend`. Similarly, renaming a workload label without updating the corresponding policy selector can leave the affected pods outside the intended isolation boundary.

This subpattern and `N-P1.1.1` may look similar but they differ in scope. In `N-P1.1.1`, no pod in the namespace is selected by an effective isolation policy, whereas in `N-P1.1.3`, some pods are isolated while others remain non-isolated.

✗ **Vulnerable manifest** (triggers N-P1.1.3)

```
# The default-deny policy selects only app=web; a pod without that label
# (e.g. app=worker) is selected by NO policy and stays fully open.
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: deny-web-only
  namespace: prod
spec:
  podSelector:
    matchLabels: { app: web }   # <-- non-empty selector leaves other pods open
        (N-P1.1.3)
  policyTypes: ["Ingress", "Egress"]
```

Remediation. Subpatterns `N-P1.1.1`–`N-P1.1.3` share a single remediation strategy. Kubernetes `NetworkPolicy` follows an allow-list model so the API does not provide an explicit deny rule. Instead, denial is implicit once a pod is selected by a policy for a given traffic direction. For that direction, the pod accepts only traffic allowed by the applicable policies and denies traffic not explicitly permitted.

A namespace-wide default-deny baseline exploits this behaviour. An empty `podSelector` selects every pod in the namespace and when both `Ingress` and `Egress` are specified in `policyTypes` and no corresponding allow rules are defined, both directions are isolated. The namespace therefore starts from a deny-by-default posture, while subsequent allow policies can only add permitted traffic to the set already allowed by the baseline.

Four further points determine whether the baseline works correctly in practice.

- **Future pods, covered automatically**: the same empty `podSelector` that establishes the baseline also matches every pod created later in the namespace. This closes the coverage gap described by `N-P1.1.3` without requiring a new policy for each workload.
- **Both directions named in `policyTypes`**: explicitly including `Egress` ensures that the egress direction is isolated as well as ingress. A policy that only establishes ingress isolation does not restrict egress merely because egress rules are absent. A generator such as the Kyverno rule associated with entry #23 addresses this by creating the baseline with both directions specified.
- **DNS allowed explicitly**: a default-deny egress baseline also prevents DNS traffic unless it is explicitly allowed by another policy. Applications that rely on cluster DNS therefore require an additional egress allow rule for the appropriate DNS service, typically in `kube-system`, on port 53. Without such an exception, workloads may lose name-resolution functionality after the baseline is introduced.
- **Present in every namespace, including future ones**: because a `NetworkPolicy` is a namespaced object, the baseline must be created in every namespace where isolation is required. Ensuring this automatically across newly created namespaces is therefore a separate operational concern, which `N-P1.2` addresses directly. Cluster-wide mechanisms discussed in Appendix A provide a different way of achieving broader coverage.

Enforcement policy

```yaml
# Default-deny baseline: empty podSelector selects ALL pods; with no ingress/
# egress rule, both directions are denied until explicitly allowed.
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny-all
  namespace: prod
spec:
  podSelector: {}                 # selects every pod in the namespace
  policyTypes: ["Ingress", "Egress"]
  # no ingress/egress stanzas -> nothing is allowed (default-deny both ways)
```

6.2.2 N-P1.2 — Missing baseline on new namespaces

Pattern definition. A lifecycle variant of missing isolation as in this case the default-deny baseline is applied to existing namespaces but not automatically to namespaces created afterwards.

Every new namespace is therefore born allow-all until someone remembers to add a baseline (a gap that widens as a cluster grows and namespaces are provisioned by CI/CD). The defect is organisational as much as technical as there is no mechanism guaranteeing that every namespace carries a baseline.

Detection criterion — N-P1.2

There exists a namespace with no default-deny `NetworkPolicy` and the cluster has no admission time or generation mechanism (policy generator, mutating control) that guarantees a baseline in every namespace.

Collected entries. Two entries, both targeting the universality of the baseline rather than a single namespace (hence a pattern distinct from `N-P1.1`):

Entries	Detection (source)
#24	all namespaces must have a NetworkPolicy (Kubescape C-0206)
#31	require / generate NetworkPolicy (Kyverno [17])

Remediation. Make the baseline automatic rather than manual by generating a default-deny `NetworkPolicy` on namespace creation, for example through the Kyverno generate rule of entry #31, so universality is enforced by the control plane and cannot be forgotten as new namespaces appear. A policy check such as the Kubescape rule of entry #24, which verifies that every namespace carries a NetworkPolicy, complements the generator by catching a namespace that was created before the automation was in place or that bypassed it. Because namespaces in a growing cluster are often provisioned by CI/CD pipelines rather than by hand, the generation step has to sit at the same admission point those pipelines go through, or the gap this pattern describes simply reappears.

Enforcement policy

```
# Kyverno (entry #31): auto-generate a default-deny baseline in every new
# namespace, so isolation universality is guaranteed at creation time.
apiVersion: kyverno.io/v1
kind: ClusterPolicy
metadata:
  name: require-default-deny
spec:
  rules:
    - name: generate-default-deny
      match:
        any:
          - resources: { kinds: ["Namespace"] }
      generate:
        apiVersion: networking.k8s.io/v1
        kind: NetworkPolicy
        name: default-deny-all
        namespace: "{{ request.object.metadata.name }}"
        synchronize: true
        data:
```

```
        spec:
          podSelector: {}
          policyTypes: ["Ingress", "Egress"]
```

6.3 Family N-F2 — Over-permissive Connectivity

Nine distinct entries are assigned to this family across four patterns. Two of them (N-P2.2, N-P2.3) are stated on analytical grounds, without a supporting detection rule.

Effect. An attacker controlling a selected pod can reach the destinations admitted by the permissive rule, enabling lateral movement within the cluster. Where the rule permits traffic to pods in other namespaces, the same lack of restriction extends the reachable scope beyond the originating namespace (ImpactScope: namespace).

Family definition

Family **N-F2** covers the misconfigurations in which a `NetworkPolicy` is present and does select the pod, but its allow-rules grant more connectivity than the workload needs. While **N-F1** is the absence of isolation, **N-F2** is isolation drawn in the wrong place. The boundary exists and it is wider than the one intended.

Threat narrative

An attacker who lands on a pod selected by an over-permissive policy reaches whatever that policy admits, and the corpus shows how wide that can be. The selector alone can span every pod of every namespace when it is left empty (`N-P2.1`), egress can reach the whole Internet when the `ipBlock` is unbounded (`N-P2.2`), and combining two selectors as a union can add one unintended namespace to the reach (`N-P2.3`). The objectives are the same as in **N-F1**, lateral movement and exfiltration. What changes is where the fix goes, because there a policy has to be written while here one has to be narrowed and narrowing it means knowing which connections the workload actually uses.

MITRE ATT&CK mapping. In MITRE ATT&CK, every pattern in the family enables lateral movement and the egress-side cases (`N-P2.2`, `N-P2.4`) enable exfiltration as well.

Technique	Why it applies
Lateral Movement *tactic*	any over-permissive rule widens what the attacker can reach
Exfiltration Over C2 Channel `T1041`	unbounded egress or open ports let data leave the cluster
Exfiltration Over Alternative Protocol `T1048`	the same egress-side cases enable this channel too

General detection criterion

Detection criterion — family N-F2

There exists a `NetworkPolicy` allow-rule whose admitted peer set (pods, namespaces, or CIDRs) and/or port set is strictly larger than the least-connectivity required by the selected workload's legitimate function.

Patterns of family N-F2

Pattern	Name	L3 variants	Entries
`N-P2.1`	Over-broad selector	`N-P2.1.1` Unrestricted peer `N-P2.1.2` Weak podSelector labels `N-P2.1.3` Over-broad namespaceSelector `N-P2.1.4` Trust on mutable labels	8

`N-P2.2`	Unrestricted ipBlock `0.0.0.0/0`	`N-P2.2.1` Egress `0.0.0.0/0`
		`N-P2.2.2` Ingress `0.0.0.0/0`
		`N-P2.2.3` Missing `except`
			0 †

`N-P2.3`	Wrong selector composition	—	0 †
`N-P2.4`	Over-permissive port exposure	`N-P2.4.1` Unrestricted ports `N-P2.4.2` Over-broad protocol `N-P2.4.3` `endPort` range too wide `N-P2.4.4` Named-port mismatch `N-P2.4.5` Unconstrained DNS egress	1

† stated on analytical grounds; no supporting detector.

6.3.1 N-P2.1 — Over-broad selector

Pattern definition. An allow-rule admits more peers than the workload needs.

An ingress rule with no `from` field places no restriction on the source of the traffic, while an egress rule with no `to` field places no restriction on its destination. A `podSelector:` used as a peer selects every pod in the policy's namespace, whereas a `namespaceSelector:` selects every namespace and can therefore match pods across the cluster. Beyond these empty forms, a selector can use labels that are too generic for the boundary it is meant to draw, or labels whose values can be set by the source itself.

Detection criterion — N-P2.1

An allow-rule matches this pattern when either (i) its `from` or `to` block is absent, empty, or carries a peer with an empty `podSelector` (`N-P2.1.1`), (ii) a peer carries an empty `namespaceSelector` (`N-P2.1.3`), (iii) a selector matches a set of peers materially larger than the workload's documented dependencies (`N-P2.1.2`), or (iv) a selector is scoped to a label the source can assign to itself (`N-P2.1.4`). The `podSelector` at the top of the specification is not part of this criterion, since an empty value there selects the pods the policy applies to, which is what a default-deny baseline requires (`N-P1.1`).

Real-world evidence (CVE)

Over-broad selectors are usually a configuration weakness without a CVE, yet two documented cases confirm the risk. `CVE-2026-32768`[42] (CVSS 7.9) concerns a miswritten `NetworkPolicy` that allows a malicious actor to pivot from an instance to any pod outside its origin namespace, breaking the expected namespace isolation and enabling potential cross-namespace lateral movement. It provides direct evidence of the over-broad connectivity described by this family, and it is assigned to `N-P2.1.3` only where inspection of the affected policy confirms that the underlying selector is an unrestricted `namespaceSelector`.
`CVE-2023-39347`[33] documents a different mechanism leading to the same class of policy failure. An attacker able to update pod labels can cause Cilium to apply incorrect network policies because the policy selection logic trusts user-provided labels during pod updates. In particular, attacker-controlled values can cause applicable policies to be bypassed. This directly supports `N-P2.1.4`, where mutable labels are trusted as selector inputs, turning a subpattern that would otherwise be analytical into one with direct vulnerability evidence.

Collected entries. Eight entries, all covering `N-P2.1.1` (the empty / allow-all case). The remaining L3 are stated on analytical grounds:

Entries	Detection (source)
#57, #61, #62, #63	allow-all recipes, empty peer (network-policy-api [16])
#65, #71	allow-from recipes, empty peer (NetworkPolicy recipes [4])
#15, #16	empty-podSelector restriction, CEL + non-CEL (Kyverno [17])

L3 — N-P2.1.1 Unrestricted peer

The broadest case of the pattern. This misconfiguration describe an ingress rule with an empty or missing `from` field or an egress rule with an empty or missing `to` field which imposes no peer restriction in that direction. The policy is present, but for that direction it does not constrain which sources or destinations the selected pods may reach. The resulting connectivity can therefore include pods in other namespaces and, where applicable, external destinations.

A further distinction is between the `ingress` or `egress` list itself and a peer written inside it. An empty or absent list for a direction named in `policyTypes`, such as `ingress: []`, leaves no rule to match against, so that direction is denied, which is the mechanism `N-P1.1`'s baseline relies on. A list holding one empty peer, such as `ingress: -{}`, does the opposite, since it is a rule that matches every peer, which is what this subpattern flags.

The empty selector to distinguish is the one written inside a peer. An empty `podSelector` inside `from` or `to` selects all pods in the policy's namespace. By contrast, an empty `podSelector` at the top level of the specification selects all pods in that namespace as the targets to which the policy applies. This is why the latter form is also used to implement a namespace-wide default-deny baseline.

✗ **Vulnerable manifest** (triggers N-P2.1.1)

```yaml
# Empty 'from' ([{}]) imposes no restriction on ingress peers.
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-all-ingress
  namespace: prod
spec:
  podSelector: {}
  policyTypes: ["Ingress"]
  ingress:
    - {}                  # <-- empty peer = allow from anywhere (N-P2.1.1)
```

L3 — N-P2.1.2 Weak podSelector labels

The `podSelector` is not empty, but it relies on a label whose value is too generic, causing the policy to select a broader set of pods than the intended isolation scope. `tier:app` is a typical example when the same label is shared by multiple otherwise unrelated workloads within the namespace. The label itself is valid and correctly identifies the selected pods but its defect lies in the breadth of the selector rather than in the authenticity or correctness of the label. This distinguishes the pattern from `N-P3.1.1`, where the selector is ineffective because the expected label is absent or does not match the target pods.

✗ **Vulnerable manifest** (triggers N-P2.1.2)

```yaml
# 'tier: app' is carried by almost every pod: the peer is far wider than the
# single client the author meant to allow.
ingress:
  - from:
      - podSelector:
          matchLabels: { tier: app }   # <-- over-broad shared label (N-P2.1.2)
```

L3 — N-P2.1.3 Over-broad namespaceSelector

A peer with an empty `namespaceSelector:` matches all namespaces and the effective scope of the peer then depends on the accompanying `podSelector`: if it is also empty, the combination matches every pod in every namespace. If it specifies labels, those labels restrict the selected pods but the restriction still applies across all namespaces. The defect is therefore the excessive namespace scope introduced by the selector, rather than the complete absence of a restrictive peer definition that characterises `N-P2.1.1`.

✗ **Vulnerable manifest** (triggers N-P2.1.3)

```yaml
# namespaceSelector: {} matches ALL namespaces.
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-from-all-namespaces
  namespace: prod
spec:
  podSelector: { matchLabels: { app: api } }
  policyTypes: ["Ingress"]
  ingress:
    - from:
        - namespaceSelector: {}   # <-- ALL namespaces (N-P2.1.3)
```

L3 — N-P2.1.4 Trust on mutable labels

The rule selects its peers by a label that the source can set on itself, such as the pod label `role:frontend`, or a namespace label the tenant is free to edit. A tenant able to label its own objects can therefore grant itself access to a workload it was never meant to reach. Here the defect is authenticity, not breadth as the selector may name a single, perfectly specific label and still be worthless because nothing checks who is entitled to carry it.

✗ VULNERABLE MANIFEST (triggers N-P2.1.4)

```
# 'role: frontend' is a label the tenant can assign to its own pod, so it can
# opt itself into this allow-rule (privilege by self-labelling).
ingress:
  - from:
      - podSelector:
          matchLabels: { role: frontend }   # <-- attacker-settable label (N-P2
              .1.4)
```

Remediation. Subpatterns `N-P2.1.1`–`N-P2.1.4` narrow the selector along different axes, so each needs its own fix.

- **Unrestricted peer** (`N-P2.1.1`): replace the empty peer with an explicit `podSelector` or `namespaceSelector` rule naming the required sources. The `podSelector` at the top of the specification is a different field and should stay empty, since that is how a default-deny baseline covers a whole namespace. Each direction, ingress and egress, needs its own explicit rule, since narrowing one leaves the pattern present on the other.
- **Weak labels** (`N-P2.1.2`): scope the selector to a label unique to the intended source, for example `app: client`, rather than a shared or generic one such as `tier: app` that nearly every workload carries. Checking the selector against how many pods it actually matches, not just its syntax, is what catches this before the rule ships.
- **Over-broad** `namespaceSelector` (`N-P2.1.3`): name the namespace through the immutable `kubernetes.io/metadata.name` label, which a tenant cannot change, rather than any other namespace label an administrator could edit. Where several namespaces have to be admitted, list them with a `matchExpressions` clause using `In`.
- **Trust on mutable labels** (`N-P2.1.4`): never authorise on a label the source can assign to itself. Use governance-controlled or immutable labels instead, the same `kubernetes.io/metadata.name` for namespaces, or a label set by a controller or admission policy for pods. `CVE-2023-39347` shows what happens when this rule is skipped, since Cilium trusted a label the pod owner could update and applied the wrong policy as a direct result.

The examples below correct each subpattern separately, since a single generic listing would not show which field changes for each.

ENFORCEMENT POLICY

```
# Corrected N-P2.1.1: name the exact source instead of an empty peer.
ingress:
  - from:
```

```
      - podSelector:
          matchLabels: { app: frontend }   # only the frontend may reach this
              pod
```

Enforcement policy

```
# Corrected N-P2.1.2: the selector is scoped to a label unique to the
# intended source, not the 'tier: app' label almost every workload carries.
ingress:
  - from:
      - podSelector:
          matchLabels: { app: client }   # unique to the intended source
```

Enforcement policy

```
# Corrected N-P2.1.3: scope to one named namespace via the immutable name
# label.
ingress:
  - from:
      - namespaceSelector:
          matchLabels:
            kubernetes.io/metadata.name: team-blue   # immutable, single
                namespace
```

Enforcement policy

```
# Corrected N-P2.1.4: the mutable pod label is paired with the immutable
# namespace identity, so a workload cannot opt itself in from a namespace
# it does not control.
ingress:
  - from:
      - namespaceSelector:
          matchLabels:
            kubernetes.io/metadata.name: team-frontend
        podSelector:
          matchLabels: { role: frontend }   # trusted only inside that namespace
```

6.3.2 N-P2.2 — Unrestricted ipBlock `0.0.0.0/0`

Pattern definition. An `egress` (or `ingress`) `ipBlock` of `0.0.0.0/0` (or `::/0`) written with no `except` clause covers the entire IPv4 (or IPv6) address space allowing traffic to and from the entire Internet and with it the cloud metadata endpoint and the internal CIDRs nested in that range. It is the IP-based counterpart of an unrestricted peer and a classic exfiltration channel.

Detection criterion — N-P2.2

An allow-rule contains an `ipBlock` with `cidr: 0.0.0.0/0` (or `::/0`) and no `except` restricting private/link-local ranges.

Collected entries. None. No surveyed tool flags an unrestricted `ipBlock` `0.0.0.0/0` or a missing `except` clause, so the pattern is stated on analytical grounds (as the theoretical RBAC subpatterns are). Three L3 by direction and refinement: `N-P2.2.1` egress, `N-P2.2.2` ingress, `N-P2.2.3` missing `except`.

L3 — N-P2.2.1 Egress `0.0.0.0/0`

The pod may open outbound connections to the entire Internet (the canonical exfiltration and command-and-control channel. Overlaps in effect with **N-F4** when the range also covers the metadata IP).

✗ **Vulnerable manifest** (triggers N-P2.2.1)

```
# Egress to 0.0.0.0/0: the pod may reach the whole Internet (exfiltration / C2).
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-egress-anywhere
  namespace: prod
spec:
  podSelector: {}
  policyTypes: ["Egress"]
  egress:
    - to:
        - ipBlock:
            cidr: 0.0.0.0/0       # <-- entire Internet in egress (N-P2.2.1)
```

L3 — N-P2.2.2 Ingress `0.0.0.0/0`

The same defect on the ingress side. An `ipBlock` of `0.0.0.0/0` placed in a `from` admits connections from any address, so the workload is open to the whole Internet.

✗ **Vulnerable manifest** (triggers N-P2.2.2)

```
# Ingress from 0.0.0.0/0: the workload accepts connections from any source IP.
ingress:
  - from:
      - ipBlock:
          cidr: 0.0.0.0/0         # <-- exposed to the entire Internet (N-P2.2.2)
    ports:
      - protocol: TCP
        port: 8080
```

L3 — N-P2.2.3 Missing `except`

The defect here is not the width of the CIDR, which is `N-P2.2.1` or `N-P2.2.2`, but the absence of an `except` list. A range meant to cover the public Internet also contains the private and link-local addresses used inside the cluster and on the node, among them the cloud metadata endpoint `169.254.169.254` that **N-F4** deals with. Writing the range without an `except` therefore leaves those internal addresses reachable and that is why the omission is a subpattern of its own. No surveyed tool checks it, so the criterion is proposed here on analytical grounds.

✗ **Vulnerable manifest** (triggers N-P2.2.3)

```
# A broad CIDR with no 'except' still reaches internal and link-local ranges
# (e.g. 169.254.169.254 metadata) -> see N-F4.
egress:
  - to:
      - ipBlock:
          cidr: 0.0.0.0/0
          # <-- no 'except' for 10/8, 172.16/12, 192.168/16, 169.254/16 (N-P2
              .2.3)
```

Remediation. Subpatterns `N-P2.2.1`–`N-P2.2.3` share one remediation. The `ipBlock` should be scoped to the address ranges required by the intended communication. When a broad CIDR such as `0.0.0.0/0` or `::/0` is necessary, the `except` field should exclude internal or sensitive address ranges that are outside the intended scope, including private or link-local networks where applicable. This reduces the possibility that the rule provides unintended access to internal services or commonly used cloud metadata endpoints. The same principle applies to ingress:

an unrestricted `0.0.0.0/0` source should be used only when traffic from arbitrary IPv4 sources is intended, and the rule should be further restricted by `ports` when only specific services need to be reachable.

Enforcement policy

```
# Corrected egress: bounded CIDR with 'except' for internal/link-local ranges.
egress:
  - to:
      - ipBlock:
          cidr: 0.0.0.0/0
          except:
            - 10.0.0.0/8
            - 172.16.0.0/12
            - 192.168.0.0/16
            - 169.254.0.0/16     # cloud metadata / link-local
```

Enforcement policy

```
# Corrected ingress: the same 'except' discipline applied to a 0.0.0.0/0
# source, plus a port restriction bounding what that open range can reach.
ingress:
  - from:
      - ipBlock:
          cidr: 0.0.0.0/0
          except:
            - 10.0.0.0/8
            - 172.16.0.0/12
            - 192.168.0.0/16
            - 169.254.0.0/16     # cloud metadata / link-local
    ports:
      - protocol: TCP
        port: 8080               # only the service port is reachable
```

6.3.3 N-P2.3 — Wrong selector composition

Pattern definition. The mistake is in how `podSelector` and `namespaceSelector` are combined inside a rule. Written in the same `from` or `to` element they are read as an "and", so the rule admits pods with label X that are in namespaces with label Y. Written as two separate elements of the list they are read as an "or", so the rule admits pods with label X in any namespace and every pod of the namespaces with label Y as well. The second form is far wider than the first and in the manifest the two differ by a single dash.

Detection criterion — N-P2.3

A `from`/`to` list contains a `namespaceSelector` and a `podSelector` as separate elements, where a single element carrying both was intended, so the rule represents a union instead of an intersection.

Collected entries. None. The union and intersection trap is described in the Kubernetes documentation but is not encoded by any surveyed scanner, so the pattern is stated on analytical grounds. The criterion above is also the weakest of the chapter, because the two forms are both valid YAML and only the author knows which one was meant in one case or another. What a tool can flag is the union form itself, but a reviewer still has to judge case by case.

✗ **Vulnerable manifest** (triggers N-P2.3)

```
# WRONG: separate list elements => UNION. This admits (a) pods app=client in ANY
# namespace, OR (b) ALL pods in namespaces team=blue. The author meant the AND.
ingress:
  - from:
      - namespaceSelector: { matchLabels: { team: blue } }
      - podSelector: { matchLabels: { app: client } }   # <-- separate => OR (N-
        P2.3)
```

Remediation. When the intent is "these pods in those namespaces", write both selectors inside the same `from` or `to` element. Separate elements should be kept for the cases where a union is really what is wanted.

Enforcement policy

```
# CORRECT: one element with both selectors => INTERSECTION (AND).
# Only pods app=client that live in namespaces team=blue are admitted.
ingress:
  - from:
      - namespaceSelector: { matchLabels: { team: blue } }
        podSelector: { matchLabels: { app: client } }    # same element => AND
```

6.3.4 N-P2.4 — Over-permissive port exposure

Pattern definition. The peer may be correctly scoped, but the `ports` constraint is broader than required by the intended communication,the rule can therefore permit traffic to ports or protocols that are not necessary for the selected workload. The block can be missing entirely, name a wider protocol or `endPort` range than the workload needs, mismatch a named port against what the target actually declares, or, on the egress side, leave DNS resolution open to any destination instead of just the cluster resolver.

Detection criterion — N-P2.4

An allow-rule matches this pattern when either (i) its `ports` block is omitted (`N-P2.4.1`), (ii) both TCP and UDP are opened on the same port where the workload needs only one (`N-P2.4.2`), (iii) a `port`/`endPort` range reaches wider than the destination exposes (`N-P2.4.3`), (iv) a named port does not match the target's `containerPort` (`N-P2.4.4`), or (v) DNS egress on port 53 is opened without restricting the destination peer (`N-P2.4.5`).

A rule is considered for `N-P2.4` when the allowed traffic is broader than required by the intended communication, for example when: (i) the `ports` field is omitted, thereby allowing all ports; (ii) the rule permits a protocol that is not required by the intended service; (iii) an `endPort` range is substantially wider than the required port range; (iv) a named port resolves to a port different from the one intended by the policy; or (v) DNS traffic is allowed without restricting the destination.

Collected entries. One entry. The other L3 are stated on analytical grounds:

Entries	Detection (source)
#45	wide port range (network-policy-api) — `N-P2.4.3`

L3 — N-P2.4.1 Unrestricted ports

The `ports` field is omitted from an allow rule, which permits traffic to all ports for the selected peer. This becomes a misconfiguration when the intended communication is limited to one or more specific service ports.

✗ **Vulnerable manifest** (triggers N-P2.4.1)

```
# No 'ports' stanza: the admitted peer reaches ALL ports of the target.
ingress:
  - from:
      - podSelector: { matchLabels: { app: client } }
    # <-- no 'ports': every port is allowed (N-P2.4.1)
```

L3 — N-P2.4.2 Over-broad protocol

The rule permits a `protocol` that is not required by the intended communication. For example, allowing both TCP and UDP when the destination service requires only TCP unnecessarily broadens the permitted traffic.

✗ **Vulnerable manifest** (triggers N-P2.4.2)

```
# UDP opened next to a TCP-only service: the UDP rule is unnecessary surface.
    ports:
      - protocol: TCP
        port: 5432
      - protocol: UDP
        port: 5432        # <-- UDP not needed by the service (N-P2.4.2)
```

L3 — N-P2.4.3 `endPort` range too wide

An `endPort` opens a contiguous segment `port-endPort` range far wider than the single service port the workload exposes, so the admitted peer receives a range where it needed one number.

✗ **Vulnerable manifest** (triggers N-P2.4.3)

```
# endPort opens the whole 1-65535 range to the admitted peer: far more than the
# single service port the workload exposes.
    ports:
      - protocol: TCP
        port: 1
        endPort: 65535    # <-- entire TCP port range (N-P2.4.3)
```

L3 — N-P2.4.4 Named-port mismatch

A rule can name a port instead of giving its number and the name is resolved against the ports declared by the pods that receive the traffic. If no pod declares that name, the rule matches nothing and grants less than intended, which is closer to what N-F3 collects than to this family. If the name is declared but maps to a different number than the author expected, the rule opens a port that was not meant to be open. The subpattern is kept here because its cause is a port declaration that no longer matches the workload, which is what this pattern is about.

✗ **Vulnerable manifest** (triggers N-P2.4.4)

```
# The rule allows port name 'metrics', but the target container exposes 'http'
# only: the named port does not match the containerPort (N-P2.4.4).
    ports:
      - protocol: TCP
```

```
        port: metrics    # <-- no containerPort named 'metrics' on the target
```

L3 — N-P2.4.5 Unconstrained DNS egress

An egress rule opens port 53 on UDP and TCP without naming a `to` peer, so DNS traffic can leave towards any destination. Name resolution works, which is the reason the rule exists, but the pod can also reach a resolver of its own choice and hide data inside the queries it sends. A `NetworkPolicy` cannot select a peer by domain name, so opening the port to everyone is a common shortcut. The defect is the missing peer, since the cluster resolver can always be named explicitly.

An egress rule permitting DNS traffic without restricting the destination allows DNS queries to any reachable DNS endpoint. In environments where DNS traffic is used for data exfiltration, such a rule can provide an additional channel for exfiltration.

✗ VULNERABLE MANIFEST (triggers N-P2.4.5)

```
# Port 53 opened with no 'to' peer: DNS resolves, but egress on 53 reaches ANY
# destination -> covert DNS exfiltration channel (N-P2.4.5).
egress:
  - ports:
      - protocol: UDP
        port: 53
      - protocol: TCP
        port: 53
    # <-- no 'to': DNS egress is unconstrained (N-P2.4.5)
```

Remediation. Subpatterns `N-P2.4.1`-`N-P2.4.5` share the same underlying fix. Every allow-rule should carry a `ports` block that lists only the ports the service exposes, with the protocol stated explicitly.

- **Ranges and named ports**: a range written with `endPort` should be used only where the workload really listens on a contiguous range, and a named port should be checked against the port names the target declares.
- **DNS** (`N-P2.4.5`): the rule needs a `to` peer as well, so that queries can only reach the cluster resolver.
- **External domains**: where traffic has to be allowed towards an external domain, some CNIs can express the destination by name, which a standard `NetworkPolicy` cannot do.

The examples below correct three of the subpatterns separately, since a single generic listing would not show which field changes for each.

ENFORCEMENT POLICY

```
# Corrected: a single, explicit port with the right protocol.
    ports:
      - protocol: TCP
        port: 5432          # only the service port, nothing else
```

ENFORCEMENT POLICY

```
# Corrected N-P2.4.3: endPort bounded to the small contiguous range the
# workload actually listens on, not the entire 1-65535 span.
    ports:
      - protocol: TCP
        port: 9090
```

```
        endPort: 9095          # bounded range, not 1-65535
```

Enforcement policy

```
# Corrected N-P2.4.4: the port name matches a containerPort the target pod
# actually declares.
    ports:
      - protocol: TCP
        port: http             # matches the 'http' containerPort on the target
```

Enforcement policy

```
# Corrected N-P2.4.5: DNS egress bounded to the cluster resolver through a
# 'to' peer, so a query cannot reach any destination the pod chooses.
egress:
  - to:
      - namespaceSelector:
          matchLabels:
            kubernetes.io/metadata.name: kube-system
        podSelector:
          matchLabels: { k8s-app: kube-dns }
    ports:
      - protocol: UDP
        port: 53
      - protocol: TCP
        port: 53
```

6.4 Family N-F3 — Ineffective Policy

*Five distinct entries are assigned to this family across three patterns (one, `N-P3.3`, is documented for completeness without a dedicated detection rule). In this particular case the policy is neither absent (**N-F1**) nor over-permissive (**N-F2**) but just has no effect.*

Effect. The policy looks restrictive but enforces nothing, so the namespace stays open while appearing segmented (ImpactScope: namespace).

Family definition

Family N-F3 groups misconfigurations in which a `NetworkPolicy` object exists and appears restrictive, yet fails to enforce the intended control. This may occur when the policy selects no pods, when a peer selector matches no intended source or destination, or when the policy configuration does not establish isolation for the traffic direction that the author intended to control. The common characteristic is that the manifest is present but its effective scope does not correspond to the intended security boundary.

Threat narrative

The policy is there and it selects the pod, but it does not restrict anything. For the attacker this is the same situation as N-F1, because the pod can still be reached from everywhere but for whoever audits the cluster it is not the same, because a tool that looks for the presence of a policy finds one and reports the workload as covered. What makes this family distinctive is that it breaks the check as well as the connectivity because a policy is written for one set of labels, the workload is

relabelled later and nobody goes back to update the policy. The rule that selected `app=api` stays in the namespace after the deployment has become `app=backend`.

MITRE ATT&CK mapping. In MITRE ATT&CK for Containers, the capability is the same as **N-F1**, with the aggravating factor that defensive visibility is undermined.

Technique	Why it applies
Lateral Movement *tactic*	the pod stays reachable exactly as in **N-F1**
Network Service Discovery T1046	but a passing policy hides it from review

General detection criterion

Detection criterion — family N-F3

There exists a `NetworkPolicy` that produces no enforcement effect on any live pod, through either (i) a `podSelector` that matches no pod (`N-P3.1.1`), (ii) a peer selector that matches nothing (`N-P3.1.2`), or (iii) a `policyTypes` declaration that omits the direction whose rules it defines (`N-P3.2`).

Patterns of family N-F3

Pattern	Name	L3 variants	Entries
`N-P3.1`	Dangling selector	`N-P3.1.1` Dangling policy `N-P3.1.2` Dangling peer	3
`N-P3.2`	Missing `policyTypes` direction	—	2
`N-P3.3`	Misread deny semantics	—	0 †

† documented for completeness; no supporting detection rule.

6.4.1 N-P3.1 — Dangling selector

Pattern definition. The selector of a policy matches no object in the cluster and what breaks depends on where that selector was written. If it is the one at the top of the policy, the selector choosing which pods the policy governs, then the policy applies to nothing at all and the pods it was meant to cover remain as open as if it had never been written, whereas if it sits inside a peer, where it names the sources or destinations that a rule admits, the rule resolves to an empty set and the traffic that should have passed is refused instead.

Detection criterion — N-P3.1

A `NetworkPolicy` has a `podSelector` (top-level or within a peer) whose label set matches zero pods/namespaces in the cluster's current state.

Collected entries. Three entries:

Entries	Detection (source)
#55	dangling policy, no target (kube-linter [49]) — `N-P3.1.1`
#53	policy not targeting any pod (KICS [14]) — `N-P3.1.1`
#32	dangling peer podSelector (kube-linter) — `N-P3.1.2`

L3 — N-P3.1.1 Dangling policy

The `podSelector` at the top of the specification matches no pod, so the policy governs nothing and the workloads it was written for are left exactly as they were before it existed.

This is the more dangerous of the two subpatterns, because the failure produces no symptom that anyone would report, while an inventory of the namespace still lists a policy and an audit asking whether one exists is satisfied by it.

✗ **Vulnerable manifest** (triggers N-P3.1.1)

```
# The podSelector targets app=api, but no pod carries that label (the workload
# is labelled app=backend). The policy exists but governs ZERO pods.
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: deny-api-ingress
  namespace: prod
spec:
  podSelector:
    matchLabels: { app: api }     # <-- no pod has app=api: dangling (N-P3.1.1)
  policyTypes: ["Ingress"]
```

L3 — N-P3.1.2 Dangling peer

A `podSelector` or `namespaceSelector` written inside a `from` or `to` block matches no peer, so the rule that was meant to open a connection resolves to an empty set and admits nobody.

Because the policy still selects its pods, those pods remain isolated in that direction and unless another policy admits the same traffic the connection the author intended is refused while the manifest reads as though it were allowed.

✗ **Vulnerable manifest** (triggers N-P3.1.2)

```
# The policy selects the right pods, but the peer it means to admit does not
# exist: no namespace carries tier=frontend, so the allow-rule never matches.
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-from-frontend
  namespace: prod
spec:
  podSelector:
    matchLabels: { app: backend }
  policyTypes: ["Ingress"]
  ingress:
    - from:
        - namespaceSelector:
            matchLabels: { tier: frontend }   # <-- matches no namespace (N-P3
                .1.2)
```

Remediation. Subpatterns `N-P3.1.1` and `N-P3.1.2` share one remediation. Selector references should be validated against the labels and namespaces present in the intended deployment, both in CI and, where appropriate, during admission. Policies whose selectors match no intended pods, or whose peers resolve to no intended endpoints, should be flagged for review. Policy and workload labels should be kept consistent so that changes to deployments do not silently invalidate the intended security boundary.

6.4.2 N-P3.2 — Missing `policyTypes` direction

Pattern definition. The policy explicitly sets `policyTypes` to a value that does not list the direction of a rule block it actually contains (typically `policyTypes: ["Ingress"]` together with an `egress` section). Kubernetes then enforces only the listed direction and **silently ignores** the rules for the other one, so the egress the author wrote has no effect and egress stays allow-all.

This is not the same as leaving `policyTypes` out entirely. When the field is absent, Kubernetes derives it from the rules (ingress is always included and Egress whenever an `egress` section exists) so an omitted `policyTypes` is safe. The defect is specifically an **explicit** `policyTypes` that excludes a written rule's direction.

It sits next to `N-P1.1.2`, but the two differ by family. **N-F1** is the plain absence of egress control, whereas here (**N-F3**) an egress rule block is actually written and then silently dropped by an explicit `policyTypes` that omits `Egress`.

Detection criterion — N-P3.2

A `NetworkPolicy` sets `policyTypes` explicitly and it does not include the direction of a rule block the policy contains (for example an `egress` section with `policyTypes: ["Ingress"]`). A policy that omits `policyTypes` altogether is not affected, since the field is then derived from the rules present.

Collected entries. Two entries:

Entries	Detection (source)
#12	pod with misconfigured NetworkPolicy (KICS)
#23	generate a correctly-typed baseline (Kyverno [17])

✗ Vulnerable manifest (triggers N-P3.2)

```yaml
# egress rules are written, but Egress is NOT in policyTypes: the rules are
# inert and egress stays allow-all.
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: restrict-egress
  namespace: prod
spec:
  podSelector: {}
  policyTypes: ["Ingress"]        # <-- Egress missing though egress rules exist
       (N-P3.2)
  egress:
    - to:
        - podSelector: { matchLabels: { app: db } }
```

Remediation. Ensure every direction with a rule block is listed in `policyTypes`. A linter rule that cross-checks rule blocks against `policyTypes` catches this deterministically.

ENFORCEMENT POLICY

```
# Corrected: Egress is declared in policyTypes, so the egress rules are enforced
    .
spec:
  podSelector: {}
  policyTypes: ["Ingress", "Egress"]   # <-- both directions now enforced
  egress:
    - to:
        - podSelector: { matchLabels: { app: db } }
```

6.4.3 N-P3.3 — Misread deny semantics

Pattern definition. A standard `NetworkPolicy` has no way of denying anything, because every rule it contains can only add traffic to what is already permitted and the only mechanism that produces a denial is selection, since a pod selected by some policy becomes closed in the directions its `policyTypes` names and stays open in all the others. Someone used to firewalls, where rules are read in order and a later one can override an earlier one, tends to write a narrow policy believing that it restricts a pod which was open before, or to add an allow-rule expecting it to take away access granted somewhere else and since neither of these things happens the policy ends up not enforcing what its author had in mind. The defect lies in the way the model was understood rather than in any field of the manifest, which is why no scanner reports it and why the pattern is documented here for completeness.

DETECTION CRITERION — N-P3.3

A policy is written as though one of its rules could deny traffic, or as though adding an allow-rule could narrow access that another policy already grants, so that the restriction its author intended is never produced.

Collected entries. None, since no surveyed tool encodes the pattern. It rests instead on the Kubernetes networking documentation, which states the additive semantics explicitly and it is retained for the same reason as the theoretical RBAC subpatterns, because it describes a failure that is common in practice and invisible to a static check.

Remediation. The only way to restrict a pod is to select it and then to add the rules naming the traffic it genuinely needs. Selection is what closes the listed directions, so it has to come first, with the allow-rules added only afterwards. In practice this means writing the default-deny baseline of **N-F1** first and treating every policy added afterwards as something that can only widen what is allowed, never as something able to take reachability away.

6.5 Family N-F4 — Sensitive Endpoint Exposure

Two distinct entries are assigned to this family, both targeting the cloud metadata endpoint. A small but high-severity family because it is about which destination a pod can reach, not how broadly.

Effect. A sensitive network endpoint, such as a cloud metadata service or a node-local Kubernetes endpoint, is reachable from a workload pod that does not require such access. The impact depends on the endpoint and its configuration: metadata exposure may provide access to cloud instance credentials, while access to node-local control-plane services may expose privileged cluster interfaces (ImpactScope: namespace).

Family definition

Family N-F4 collects misconfigurations in which a workload pod can reach a specific sensitive endpoint that should not be part of its network reachability. Examples include cloud instance metadata services and node-local Kubernetes control-plane endpoints such as the kubelet. Unlike N-F2, where the defect is primarily the excessive breadth of an allow-rule, N-F4 focuses on the exposure of a specific sensitive destination. The security relevance therefore depends on the properties of the endpoint itself rather than only on the size of the permitted address range.

The family has a single pattern, `N-P4.1` (Sensitive endpoint reachable), subdivided by the endpoint class.

Threat narrative

The canonical case is the cloud instance metadata service, commonly exposed through `169.254.169.254`. Depending on the cloud platform and instance configuration, the metadata service may expose information or temporary credentials associated with the workload's host or instance role. A pod that can reach the service may therefore obtain information or credentials that were not intended to be available to the workload and potentially use them against cloud resources accessible to that identity.

MITRE ATT&CK mapping. In MITRE ATT&CK, the family maps to the following techniques.

T1528 Steal Application Access Token → "reaching node-local endpoints (kubelet, etcd) steals their access tokens"

Technique	Why it applies
Unsecured Credentials: Cloud Instance Metadata API `T1552.005`	a pod reaching the metadata service can retrieve the node's IAM credentials
Steal Application Access Token `T1528`	reaching node-local endpoints (kubelet, etcd) steals their access tokens

General detection criterion

Detection criterion — family N-F4

A workload pod's egress is permitted (by an explicit rule, or by the absence of egress isolation) to reach a sensitive endpoint (as the metadata IP `169.254.169.254`, or a node local control plane port such as `10250`/`2379`) that its function does not require.

6.5.1 N-P4.1 — Sensitive endpoint reachable

Pattern definition. A pod can reach an endpoint that no ordinary workload should, and the pattern distinguishes two such endpoints: the cloud metadata service (`N-P4.1.1`) and a control plane port listening on the node itself (`N-P4.1.2`), such as the kubelet. Only the metadata case appears in the corpus while the node-local one is added for completeness, and corresponds, on the network side, to the substrate access `R-P4.6` describes for RBAC.

Detection criterion — N-P4.1

A workload pod remains network-reachable to a sensitive endpoint because the applicable egress policies do not exclude that destination from the effective allow set. The endpoint classes are (i) the cloud metadata address `169.254.169.254` (`N-P4.1.1`) and (ii) node-local Kubernetes endpoints such as the kubelet on `10250` and, where applicable, etcd on `2379` (`N-P4.1.2`).

Collected entries. Two entries, both `N-P4.1.1`. No tool isolates the node local port case `N-P4.1.2` and this is flagged here as a tooling gap:

Entries	Detection (source)
#105, #118	egress to cloud metadata / IMDS (Kubescape [3]) — `N-P4.1.1`

A note on `N-P4.1.3`. A third endpoint class (the control plane / API server reachable from a pod) was considered but excluded from the taxonomy as its detection signal and remediation coincide with those of the node local ports (`N-P4.1.2`), so it is treated as overlapping rather than as a distinct L3. It is recorded here for completeness of the endpoint space.

L3 — N-P4.1.1 Cloud metadata endpoint

The instance metadata service answers on `169.254.169.254` without authentication, from inside the node itself and returns the node's cloud IAM credentials to whatever process asks for them. If egress to that single address is not blocked, any pod able to reach it can retrieve those credentials and pivot into the cloud account.

✗ **Vulnerable manifest** (triggers N-P4.1.1)

```
# Egress permits 169.254.0.0/16: the pod can query the cloud metadata service
# at 169.254.169.254 and retrieve the node's IAM credentials.
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-egress-internal
```

```
  namespace: prod
spec:
  podSelector: {}
  policyTypes: ["Egress"]
  egress:
    - to:
        - ipBlock:
            cidr: 169.254.169.254/32    # <-- includes 169.254.169.254 metadata (
                N-P4.1.1)
```

L3 — N-P4.1.2 Node local control plane ports

The pod has egress reachability to a node-local Kubernetes endpoint outside its intended communication scope. The kubelet API on port `10250` is the primary example. Where an etcd endpoint is network-reachable from the workload network, access to its client port (`2379`) should likewise be treated as sensitive.

✗ Vulnerable manifest (triggers N-P4.1.2)

```
# Egress is allowed to the node subnet with no port restriction: the pod can
# reach the kubelet on 10250 and etcd on 2379 on every node.
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-egress-nodes
  namespace: prod
spec:
  podSelector: {}
  policyTypes: ["Egress"]
  egress:
    - to:
        - ipBlock:                  # Example only: 10.20.0.0/24 represents the
            node subnet in this cluster.
          cidr: 10.20.0.0/24        # Without a port restriction, the rule permits
              access to node-local
                                    # services including the kubelet port where
                                        exposed.
```

Remediation. Subpatterns `N-P4.1.1` and `N-P4.1.2` share one remediation. It does not consist of an `except` carved out of a wide range. An `ipBlock` of `0.0.0.0/0` with the metadata address and the node ranges excluded is still broader than either vulnerable example above, because every other undocumented destination stays reachable under that shape. The rule should instead name the specific destinations and ports the workload requires, so that the metadata address and the node ranges are absent from the allow-list rather than merely carved out of an open one.

Enforcement policy

```
# Corrected: egress named to the destinations the workload actually needs,
# not to a wide range with exceptions carved out. Neither the metadata
# address nor the node range appears anywhere in this allow-list.
egress:
  - to:
      - ipBlock:
          cidr: 172.20.5.0/24     # the database subnet this workload talks to
    ports:
      - protocol: TCP
        port: 5432
  - to:
      - namespaceSelector:
          matchLabels:
            kubernetes.io/metadata.name: kube-system
        podSelector:
          matchLabels: { k8s-app: kube-dns }
```

```
    ports:
      - protocol: UDP
        port: 53
```

6.6 Family N-F5 — Enforcement Layer Preconditions

*12 distinct entries are assigned to this family across two patterns, six for `N-P5.1` and six for `N-P5.2`, the latter obtained from 27 raw entries once near duplicates are excluded. Like **R-F5** on the RBAC side, the family is foundational and borderline at the same time, because it concerns the substrate that NetworkPolicy depends on rather than the policy object.*

Effect. Even correct policies are not enforced, so no network isolation holds anywhere in the cluster (ImpactScope: cluster).

Family definition

Unlike the previous families, **N-F5** does not collect misconfigurations of a policy object but the conditions that have to hold before a `NetworkPolicy` can take effect at all. The object itself is only a declaration and the work of enforcing it belongs to the CNI plug-in, so a cluster whose plug-in ignores the API stores policies that never do anything.

The effectiveness of NetworkPolicy also depends on the traffic remaining within the scope that the NetworkPolicy implementation can mediate. Certain host-level networking configurations can place traffic outside the normal pod-network model or make its treatment implementation-dependent.

The family is marked BORDERLINE because `N-P5.2` belongs to Pod Security as well, given that a pod with `hostNetwork: true` is a finding for both. Following the scoping decision of Chapter 4 these entries stay here for their effect on the network, while their Pod Security character is recognised.

Threat narrative

N-F5 is the network-side counterpart of **R-F5**, since neither is a misconfiguration that can be exploited on its own and both describe the condition that cancels an entire class of controls at once. Where the CNI ignores the NetworkPolicy API, every policy in every namespace is ineffective and no review of the policies can reveal it because each one is written correctly. Where a pod runs with `hostNetwork`, it shares the network namespace of the node and its traffic is simply not pod traffic, so it reaches node-local services and the metadata endpoint without any policy having any effect.

MITRE ATT&CK mapping. Like **R-F5** (Section 5.6), **N-F5** maps to the precondition behind other techniques rather than to a technique of its own: a CNI that silently stops enforcing is not itself an adversary action, so no single technique is listed here.

General detection criterion

Detection criterion — family N-F5

Either (i) the cluster's CNI does not implement the NetworkPolicy API, so that policies have no runtime effect (`N-P5.1`), or (ii) a pod is configured through `hostNetwork` or `hostPort`, so that its traffic is not mediated by the pod-network policy plane (`N-P5.2`).

Patterns of family N-F5

Pattern	Name	L3 variants	Entries
`N-P5.1`	NetworkPolicy not enforced by CNI	`N-P5.1.1` CNI without NetworkPolicy support `N-P5.1.2` Managed-platform enforcement disabled	6
`N-P5.2`	Host-network bypass	`N-P5.2.1` Shared host network namespace `N-P5.2.2` Node-published container port `N-P5.2.3` NodePort selector bypass	6 (27)

For `N-P5.2` the first figure counts the canonical entries and the second the raw ones, 21 of which are near duplicates marked `DuplicateOf`.

6.6.1 N-P5.1 — NetworkPolicy not enforced by CNI

Pattern definition. Enforcement of NetworkPolicy is not provided by Kubernetes itself but by the plug-in installed under it and this pattern covers the cluster where that plug-in does not provide it, whether because it has no such capability or because a managed platform offers it as a feature that has been left off. In both cases the API server accepts every `NetworkPolicy` object and stores it, while nothing ever applies it.

Detection criterion — N-P5.1

The active CNI does not advertise NetworkPolicy support, or the managed cluster's network-policy add-on is not enabled.

Real-world evidence (CVE)

Unlike the configuration families N-F1–N-F4, this one is backed by real CVEs, because the substrate is software and not a configuration choice. Three were verified by hand and all three are enforcement failures on the CNI side. In `CVE-2023-27595` [29] the eBPF programs of Cilium are not yet attached at start-up, so for a short window the host enforces none of the policies. `CVE-2024-7598` [39] is a network-restriction bypass through a race condition while a namespace is terminating. In `CVE-2026-33726` [43] Cilium does not enforce ingress policies for pod-to-service traffic whose backend is on the same node. Together they show that a correct policy that is not enforced is a real and exploited failure, which is what determines the severity of `N-P5.1`.

Collected entries. Six entries:

Entries	Detection (source)
#4	CNI does not support NetworkPolicy (KICS [14])
#10, #11	require / generate policy, incl. DNS (Kyverno [17])
#28, #30	enforcement enabled on EKS (Kubescape [3])
#80	CNI enabled on AKS (Kubescape)

L3 — N-P5.1.1 CNI without NetworkPolicy support

The plug-in installed in the cluster does not implement the NetworkPolicy API at all. This is a design choice rather than a fault, since some plug-ins are built only to give pods connectivity and leave filtering to something else. The policy objects are accepted by the API server and stored and no component ever reads them.

✗ VULNERABLE MANIFEST (triggers N-P5.1.1)

```yaml
# A cluster running a CNI without NetworkPolicy support: this default-deny is
# accepted by the API server but NEVER enforced -> all traffic still flows.
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny-all
  namespace: prod
spec:
  podSelector: {}
  policyTypes: ["Ingress", "Egress"]
  # Inert unless the CNI (Calico, Cilium, Antrea, ...) enforces NetworkPolicy.
```

L3 — N-P5.1.2 Managed-platform enforcement disabled

On managed Kubernetes platforms, NetworkPolicy enforcement may depend on the networking mode and on a provider-specific network-policy feature or configuration. When that capability is disabled, NetworkPolicy objects may still be accepted by the Kubernetes API without providing the intended runtime enforcement.

Remediation. Run a CNI that implements the API, such as Calico, Cilium or Antrea and do not take the claim on trust. The check that matters is a connectivity test where a default-deny policy is applied to a namespace and a connection that used to work is verified to fail.

On managed clusters, enable and verify the provider-supported NetworkPolicy enforcement mechanism appropriate to the selected networking configuration.

6.6.2 N-P5.2 — Host-network bypass

Pattern definition. A pod leaves the pod network and NetworkPolicy, which mediates only the traffic travelling on that network, no longer applies to it. The 27 raw entries collected for this pattern reduce to six once near duplicates are excluded (Section 6.1) and many of them are Pod Security controls kept here for the effect they have on the network.

Detection criterion — N-P5.2

A pod sets `hostNetwork: true` or declares a `hostPort`, which places it directly in the node's own network namespace, structurally out of reach of the pod-network policy plane regardless of CNI, or a Service of type `NodePort` exposes a workload on the node address, where the resulting source address rewrite breaks pod/namespace-selector matching even though the traffic still lands in the pod's own network namespace and remains subject to enforcement there, CNI implementation permitting.

Collected entries. Six canonical entries out of 27, the remaining 21 being near-duplicate tool rules marked `DuplicateOf`:

Entries	Detection (source)
#79	hostNetwork access (Kubescape C-0041) — `N-P5.2.1`
#96, #100	shared host network namespace (Gatekeeper [44], Checkov [5]) — `N-P5.2.1`
#106	container hostPort (Kubescape C-0044) — `N-P5.2.2`
#69	disallow NodePort (Kyverno) — `N-P5.2.3`
#95	sensitive-ports guidance (OWASP [45])

L3 — N-P5.2.1 Shared host network namespace

The pod sets `hostNetwork: true` and shares the network namespace of the node, which makes it the most complete of the three bypasses, since its traffic is never pod traffic and no policy is ever consulted for it.

✗ **Vulnerable manifest** (triggers N-P5.2.1)

```
# hostNetwork: true -> the pod shares the NODE's network namespace and is NOT
# subject to pod NetworkPolicy. It reaches node-local services and metadata.
apiVersion: v1
kind: Pod
metadata:
  name: agent
  namespace: prod
spec:
  hostNetwork: true          # <-- bypasses the NetworkPolicy plane (N-P5.2.1)
  containers:
    - name: agent
      image: monitoring-agent:1.0
```

L3 — N-P5.2.2 Node-published container port

A container exposes a port directly on the node through `hostPort`. This places the service on a node-level address rather than exposing it solely through the pod network. The exact interaction with NetworkPolicy is implementation-dependent and therefore this pattern is treated as a host-networking boundary condition rather than as a guaranteed policy bypass.

✗ **Vulnerable manifest** (triggers N-P5.2.2)

```
# hostPort publishes the container's port directly on the node's address,
# reachable without ever passing through the pod network.
apiVersion: v1
kind: Pod
metadata:
  name: agent
  namespace: prod
spec:
  containers:
    - name: agent
      image: monitoring-agent:1.0
      ports:
        - containerPort: 9100
          hostPort: 9100      # <-- published on the node IP (N-P5.2.2)
```

L3 — N-P5.2.3 NodePort selector bypass

A Service of type `NodePort` publishes a workload on the address of every node in the cluster. What arrives at the pod then carries the address of the node rather than that of the original client, so rules written against pod or namespace selectors do not match it. Whether the traffic is filtered by anything else depends on the CNI: most enforce `ipBlock`-based or default-deny rules against it regardless, so the bypass is specific to selector-based matching rather than a wholesale exit from the policy plane.

✗ **Vulnerable manifest** (triggers N-P5.2.3)

```
# type: NodePort publishes the Service on every node's address. Traffic then
# arrives carrying the node's IP rather than the client's, so it skips
# whatever pod or namespace selector a NetworkPolicy rule was written
# against (entry #69).
apiVersion: v1
kind: Service
metadata:
  name: web
  namespace: prod
spec:
  type: NodePort            # <-- published on every node address (N-P5.2.3)
  selector:
    app: web
  ports:
    - port: 80
      targetPort: 8080
      nodePort: 30080
```

Remediation. All three subpatterns share one remediation and none of it can be written as a `NetworkPolicy`.

- **Pod level**: ordinary workloads should refuse `hostNetwork`, `hostPort` and the host namespaces through Pod Security admission at the `baseline` or `restricted` level, or through a custom admission policy where a finer rule is needed.
- **Service level**: Services should be published with a ClusterIP behind an Ingress rather than with a `NodePort`.

Chapter 7

Cross-Domain Misconfigurations

This chapter presents the cross-domain taxonomy. It first explains the method (Section 7.1–Section 7.3) and then the composition matrix and the cross-families themselves (Section 7.4 onward). Unlike the RBAC and NetworkPolicy taxonomies, which were built bottom up from detection tools and online sources, this one is built compositionally, for reasons set out below.

7.1 What a cross-misconfiguration is

The two control planes divide the work between them. RBAC governs what an identity may do on the API and NetworkPolicy governs what a pod may reach on the network, so an attack chain normally needs one thing from each side: a capability that an RBAC weakness grants and, on the NetworkPolicy side, either a gap a weakness keeps open or a control that weakness disables outright. The outcome exceeds what either side allows on its own once the two meet.

A cross-misconfiguration is defined here as a weakness that exists in neither the RBAC configuration nor the NetworkPolicy configuration taken alone and that emerges when the two are read together. It takes one of two shapes.

- **Compositions** (eight of the ten patterns), where both sides are weak and the two weaknesses have to hold together.
- **Disabling patterns** (the remaining two, `X8` and `X9`), where only the RBAC side is weak and the permission it grants disables a network control that was written correctly.

Because there is no measure for the size of a capability, neither shape can be checked by comparing sizes directly. The two criteria below state each requirement in a form that can be verified instead.

Criterion for compositions. Instead of comparing capability sizes directly, the criterion asks whether each side is necessary for the attacker's goal g.

COMPOSITION CRITERION

Writing R for the set of RBAC patterns of Chapter 5 and N for the NetworkPolicy patterns of Chapter 6, take a pair $(r, n) \in R \times N$ and an attacker goal g. The pair is a composition when g is unreachable with only r present, unreachable with

only n present and reachable once both hold together.

Each pattern already reports which single fix breaks the chain, so checking is direct. Since removing one side and seeing whether the goal still holds confirms that it was necessary, doing this for both sides is what makes the composition genuine rather than just useful. Rejecting a candidate works the same way, by finding one goal that a single side alone can already reach, which is how the borderline cells of Section 7.4 were excluded.

A cross-pattern is distinct from another when its goal g or its pair of capabilities $C(r, n)$ differs.

Criterion for disabling patterns. The composition criterion above was fixed before the matrix in Section 7.4 was examined and it is what decided which cells to keep. `X8` and `X9` came later, once the retained cells were checked one pattern at a time. Both kept looking like genuine cross-domain weaknesses while failing the symmetric test. Dropping them or stretching the first criterion to admit them would not have been honest, so a second criterion was written to describe what they actually are.

Disabling criterion

Take the RBAC pattern r behind `X8` or `X9` and an attacker goal g. The pattern is a disabling one when g is unreachable with the NetworkPolicy enforced as written and becomes reachable once the permission r grants is used against that policy.

The permission in both cases targets the NetworkPolicy object itself: the isolation was written correctly and stays enforced until that permission is used to remove it, which leaves no weak network side to fix.

The direction follows from what the two objects are, not from a choice. RBAC governs the API, so a permission can reach a NetworkPolicy as one of the objects it stores. A NetworkPolicy itself only governs packets and, since a packet cannot rewrite an object, nothing on the network side can touch RBAC the same way. The closest thing to a reversed case is a pod that reaches the kubelet directly, skipping the API server, but that only works if the kubelet authorises requests on its own without consulting RBAC (`R-P5.1.3`), which means the RBAC side is not correct there either and the case is really a composition, not a one-directional pattern.

Both shapes share one property: neither can be seen from one plane alone. A composition needs a weakness on each side, so an audit of either domain sees half of it. A disabling pattern is worse, because an RBAC audit finds a permission, a network audit finds a correct policy and **both audits pass** while the cluster is exposed.

7.2 Why this taxonomy cannot be built bottom up

Both single domain taxonomies were built bottom up, by surveying detection tools, collecting the rules they define, normalising them into a dataset and letting the patterns emerge from the data. That method is deliberately not repeated here and the data themselves explain why.

Intersecting the two datasets leaves almost nothing. Of roughly one hundred distinct CVEs referenced across both datasets, only two appear in both (`CVE-2022-46167` [27], `CVE-2024-31989` [38]) and only a single RBAC entry in the dataset so much as mentions NetworkPolicy resources. The emptiness is structural rather than an artefact of how the data were collected, because **detection tools are single domain by construction** and a scanner audits RBAC or NetworkPolicy, never their interaction. Neither the tools nor the CVE record catalogue combined weaknesses, so a fresh keyword driven collection would mostly confirm the void rather than populate a taxonomy.

The cross-domain layer is under detected precisely because no tool looks across the boundary, and this justifies the analyst driven construction used in this chapter.

7.3 Construction method: compose, formalise, validate

The cross-taxonomy is built in three steps:

Step 1 — Generation (top-down composition). The candidate space is the product of the two existing taxonomies, the five RBAC families (**R-F1**–**R-F5**, 17 patterns) against the six NetworkPolicy families (**N-F1**–**N-F6**, 15 patterns), which gives thirty cells to examine. The pairs are therefore enumerated rather than discovered and each one is kept only if it satisfies one of the two criteria of Section 7.1, either an RBAC capability and a NetworkPolicy reachability that are both needed, or an RBAC permission that disables a policy which is otherwise correct. Ten cells survive out of the thirty. The retained patterns are then organised by the **attacker capability the pair unlocks**, such as lateral movement, data exfiltration or reaching across a boundary. This is a different axis from the one used in Chapter 5 and Chapter 6, where a family is named after the security property that breaks and the change is forced by the object of study. A single domain misconfiguration breaks one property, which is what gives it exactly one family, whereas a cross-pattern breaks one property on each side by construction, so every pattern here would belong to an RBAC family and to a NetworkPolicy family at the same time and the property axis would separate nothing. What does separate them is what the pair makes possible. Defined through the joint capabilities the pair unlocks, these families align more closely with high-level attacker objectives and map onto MITRE ATT&CK tactics more directly as a result.

Step 2 — Formalisation. Each retained pattern is written up with the same schema used throughout the thesis, extended to two objects:

- **Composition**, the RBAC predicate together with the NetworkPolicy predicate, both of which have to hold.
- **Equivalent patterns**, the alternative instances, on either side, that reach the same composed capability; present for every pattern except `X10`, whose precondition has none.

- **Why this family, not another**, included only where a pattern could be mistaken for a similar-looking one and needs disambiguating.
- **Resulting capability**, the capability $C(r, n)$ that the pair unlocks.
- **Joint predicate**, a single check over both object kinds that decides whether a cluster exhibits the pattern.
- **Minimal fix**, the remediations available on each side, with an indication of which single one is enough to break the chain.
- **Validation**, what the dedicated lab in Chapter 8 found when it reproduced the pattern directly.

Step 3 — Validation and grounding. Because the patterns are generated top down, evidence comes in afterwards rather than before. Each pattern is grounded on two sources:

- **A mapping onto MITRE ATT&CK**, one table per cross-domain family (Section 7.5), an external and recognised vocabulary that ties a chain derived on paper to a framework other people already use.
- **A dedicated lab** (Chapter 8), evidence this thesis produced directly rather than borrowed from an existing framework: it reproduces each retained pattern on a running cluster, and every pattern box in Section 7.5 records what it found, including `X10`, whose entry explains why no single call could reproduce it.

While the single-domain taxonomies derive patterns inductively from empirical data, the cross-domain layer proceeds from patterns to attack chains before validating them against evidence, referring to the methodological gap highlighted in Section 7.2. With the method now set out, the rest of the chapter puts it to work. It first shows the composition matrix that builds the candidate space (Section 7.4). It then presents the cross-taxonomy itself, four cross-families, each named after the capability it unlocks and holding eight formalised cross-patterns together. A fifth, borderline family, `X10`, covers the case where both enforcement layers are missing at once. It is treated as a foundation condition, not as an attacker capability (Section 7.5).

7.4 The RBAC × NetworkPolicy composition matrix

Table 7.1 is the fundamental tool used to discuss the cross-domain layer. Rows are the RBAC families (Chapter 5) and columns the NetworkPolicy families (Chapter 6) and each cell asks whether an RBAC weakness of that family and a NetworkPolicy weakness of that family satisfy one of the two criteria already fixed in Section 7.1: the composition criterion, where the attacker's goal is unreachable with either side alone and reachable once both hold together, or the disabling criterion, where an RBAC permission acts directly on an otherwise-correct NetworkPolicy and removes it.

A cell that satisfies either one carries the identifier of the resulting cross-pattern and a cell that does not carries the reason, so that no cell is left blank. The matrix itself works at the level of families but each composing cell is instantiated by one or more specific sub-patterns, named individually in the row-by-row derivation below and in the pattern boxes of Section 7.5, with the granularity limits stated in Section 7.6.

Table 7.1: Composition matrix: a filled cell names the cross-pattern of Section 7.5, every other cell the reason it does not compose (legend below). The matrix works at family level so a cell can hold more than one sub-pattern outcome, see the declared limits in Section 7.6.

RBAC \ NetPol	**N-F1** Missing Isol.	**N-F2** Over-perm. Conn.	**N-F3** Ineffective Policy	**N-F4** Sens. Endpoint	**N-F5** Substrate	**N-F6** CNI/Admin
R-F1 Identity Overexposure	X1	X4	COLL	NET*	SUB	COLL
R-F2 Permission Overreach	COLL	COLL	X8†	NET	SUB	X9†
R-F3 Sensitive Resource Exp.	COLL→X3	X3	COLL→X3	X7	SUB	COLL
R-F4 Privilege Escalation	COLL→X5/CP	X5	COLL→X5/CP	X6	SUB	COLL→X5/CP
R-F5 Authz Preconditions	SUB	SUB	X10‡	SUB	X10‡	SUB

CP control plane RBAC NET network-only capability SUB borderline substrate COLL collapses into an existing pattern.

A slash joins two reasons in one cell when the family carries both an instance that composes and one that stays effective regardless of the network side: **R-F4** is the only such family (Section 7.4).

† disabling pattern: only RBAC is weak, removing an otherwise correct policy.

* validated in Chapter 8: a dedicated lab, a reversed-outcome check or the X2 refutation.

‡ borderline foundation condition, not an attacker capability in its own right (Section 7.5.5); counted separately from the eight cross-patterns throughout this chapter.

Of the thirty family-level cells, ten are occupied by the eight cross-patterns that each hold one cell while the borderline condition X10 recurs in the remaining two (**R-F5**×**N-F3** and **R-F5**×**N-F5**).

Analysis of the results: the five cross-domain families (**XF1**–**XF5**) were derived empirically from the composition rather than defined a priori. Section 7.3 first constructs the 30-cell candidate space top-down and only then are the valid patterns grouped according to the specific attacker capability they unlock. Importantly, this set of five capabilities reflects the empirical findings of our dataset rather than a closed taxonomy, meaning additional capabilities could emerge under a different corpus.

To ensure completeness, every cell was systematically tested against all five capabilities, not just the most intuitive match. A cell is assigned to a capability only if it satisfies the composition or disabling criteria detailed in Section 7.1. The following analysis explicitly documents both why a matching capability applies and why the remaining four are excluded.

Row R-F1 — Identity Overexposure. None of the row's six cells reaches **XF4**: an overexposed identity's grants target Kubernetes resources broadly, never a NetworkPolicy or admin object directly, which is what X8/X9 require.

Table 7.2: **R-F1** (Identity Overexposure) checked against every NetworkPolicy family.

NetPol family	Result	Why
N-F1 Missing Isol.	`X1` (**XF1**)	The identity supplies broad reach, the missing baseline supplies the open path to any peer (Section 7.5.1).
N-F2 Over-perm. Conn.	`X4` (**XF2**)	Same identity, this time paired with open egress out of the cluster (Section 7.5.2).
N-F3 Ineffective Policy	COLL → `X1`	An ineffective policy is indistinguishable from **N-F1**'s missing baseline for the attacker. It collapses rather than opening a new pattern (Table 7.7).
N-F4 Sens. Endpoint	NET*	The reversed test (Chapter 8) shows the metadata endpoint is reached whether or not the ServiceAccount is scoped down: the network side alone already provides this. **XF3** still reaches this endpoint, but through **R-F3**/**R-F4** instead (Section 7.5.3), where the RBAC side adds something.
N-F5 Substrate	SUB	An unenforcing CNI is a substrate precondition, not a capability. It degenerates into `X10` only when paired with **R-F5**'s own precondition, not **R-F1**'s.
N-F6 CNI/Admin	COLL → `X1`	This reopens the same connectivity **N-F1** lacks by default, by the same mechanism as **N-F3** (Table 7.7).

Row R-F2 — Permission Overreach. A single over-reaching rule, such as `R-P2.1.2`'s wildcard over `resources`, grants the same broad API reach as an over-exposed identity, without the binding itself being wrong.

Table 7.3: **R-F2** (Permission Overreach) checked against every NetworkPolicy family.

NetPol family	Result	Why
N-F1 Missing Isol.	COLL → `X1`	Same capability as **R-F1**×**N-F1**: only the excess moves from the whole identity into one rule (Table 7.7).
N-F2 Over-perm. Conn.	COLL → `X4`	Same collapse, this time paired with open egress (Table 7.7).

continues on next page

Table 7.3 (continued)

NetPol family	Result	Why
N-F3 Ineffective Policy	X8† (**XF4**)	A rule broad enough to reach `networkpolicies` on write verbs (`R-P2.1.2`/`R-P2.1.3`) can rewrite or remove an otherwise correct policy. **XF1**–**XF3** do not apply: nothing is composed, the RBAC side disables the network's control object directly.
N-F4 Sens. Endpoint	NET	Adds nothing beyond what **R-F1**×**N-F4** already ruled out, for the identical reason: a bare wildcard over `resources` does not place the workload where the endpoint would matter.
N-F5 Substrate	SUB	Same substrate precondition as every row (**R-F5**'s own, not **R-F2**'s).
N-F6 CNI/Admin	X9† (**XF4**)	Mirrors **N-F3**: the same kind of wildcard, this time reaching the CNI or admin-policy CRDs, disables the substrate layer directly.

Row R-F3 — Sensitive Resource Exposure. Reading a Secret (`R-P3.1`) is API-only on its own: it needs a network side to carry the content out, or to supply the endpoint it names.

Table 7.4: **R-F3** (Sensitive Resource Exposure) checked against every NetworkPolicy family.

NetPol family	Result	Why
N-F1 Missing Isol.	COLL → X3	A missing baseline defaults both directions to allow-all (`N-P1.1.1`, Section 6.2.1), so it already grants the egress X3 needs. Same capability as **R-F3**×**N-F2**, reached through the absence of a policy rather than an explicit open rule (Table 7.7).
N-F2 Over-perm. Conn.	X3 (**XF2**)	The read achieves nothing without a way out, and the egress is harmless with nothing sensitive behind it.
N-F3 Ineffective Policy	COLL → X3	An ineffective policy is indistinguishable from **N-F1**'s missing baseline for the attacker, so it collapses the same way, into X3 (Table 7.7).

continues on next page

Table 7.4 (continued)

NetPol family	Result	Why
N-F4 Sens. Endpoint	X7 (**XF3**)	The RBAC side supplies what mere reachability does not: knowledge of where the endpoint is, from reading the Endpoints/ConfigMap object that records it. Not **XF1**/**XF2** (this is not lateral movement or arbitrary exfiltration) and not **XF4**, since a read-only instance cannot disable anything.
N-F5 Substrate	SUB	Same substrate precondition as every row.
N-F6 CNI/Admin	COLL → X3	Reopens the connectivity X3 needs, same mechanism as **R-F1**×**N-F6** (Table 7.7).

Row R-F4 — Privilege Escalation. The row splits by instance: creating a pod (R-P4.4) composes with the network below, while `kubectl exec` (R-P4.5) and credential issuance (R-P4.2) never leave the API server, so neither composes with anything.

Table 7.5: **R-F4** (Privilege Escalation) checked against every NetworkPolicy family.

NetPol family	Result	Why
N-F1 Missing Isol.	COLL → X5/CP	`exec` and credential issuance stay inert here for the same reasons as every other NetPol family: `exec` is proxied through the API server to the kubelet, never pod-to-pod traffic a NetworkPolicy can see (X2, refuted, Chapter 8), and an approved CSR or minted token authenticates against the API server, not a peer, so there is nothing for the network to carry the way X1's stolen Secret is carried. R-P4.4 is different: a missing baseline defaults egress to allow-all too, the same channel N-P2.2 opens explicitly, so the created pod already satisfies X5's precondition (Table 7.7).
N-F2 Over-perm. Conn.	X5 (**XF2**)	The escalation verb supplies the workload, the open egress supplies the channel to send data out.
N-F3 Ineffective Policy	COLL → X5/CP	Same evidence as **N-F1** for `exec` and credential issuance: both bypass the network layer whether the policy in place is missing, over-permissive or ineffective. An ineffective policy is indistinguishable from a missing baseline for the attacker, so R-P4.4 collapses the same way, into X5 (Table 7.7).

continues on next page

Table 7.5 (continued)

NetPol family	Result	Why
N-F4 Sens. Endpoint	`X6` (**XF3**)	The escalation verb supplies the placement, the endpoint supplies the target: a pod built to reach one endpoint, not to move laterally or exfiltrate at will.
N-F5 Substrate	SUB	Same substrate precondition as every row.
N-F6 CNI/Admin	COLL → `X5`/CP	Checked against `X6` and rejected: `X6` needs the specific metadata/kubelet endpoint **N-F4** reaches, not the generic connectivity **N-F6** reopens, so that analogy does not carry over. `X5` needs only open egress, which an over-broad cluster-wide or admin override reopens the same way it does for **R-F1**×**N-F6** and **R-F3**×**N-F6** (Table 7.7).

Row R-F5 — Authz Preconditions. **R-F5** is a foundation family in the same sense as **N-F5**: a disabled or misconfigured authorizer is a precondition for every other weakness to matter, not an attacker capability with a target of its own.

Table 7.6: **R-F5** (Authz Preconditions) checked against every NetworkPolicy family.

NetPol family	Result	Why
N-F1, **N-F2**, **N-F4**, **N-F6**	SUB	None of **XF1**–**XF4** can be produced by this row acting alone or paired with an ordinary NetworkPolicy family.
N-F3 Ineffective Policy	`X10` (**XF5**)	Authorizer disabled while the workload's own policy still shows as present: it degenerates into the borderline capability where every other weakness becomes exploitable at once, without a distinguishing mechanism of its own.
N-F5 Substrate	`X10` (**XF5**)	Same degenerate capability, this time with the substrate itself failing to enforce (also why `X10` is analysed rather than run in the validation lab, Chapter 8).

Why the other cells do not compose. This section summarizes the three recurring exclusion mechanisms responsible for the twenty non-composing cells, expanding on the matrix outcomes presented in Table 7.1:

- **Collapse into an existing pattern** (ten cells, marked COLL): Each cell reduces, at the sub-pattern level, to an instance already captured by an existing cross-pattern rather than introducing a new capability (Table 7.7). These ten collapses are driven by three distinct mechanisms:

1. **RBAC-side substitution:** an over-reaching **R-F2** rule grants the same broad scope as an overexposed **R-F1** identity. Consequently, **R-F2**×**N-F1** collapses into `X1` and **R-F2**×**N-F2** collapses into `X4`, directly mirroring their **R-F1** counterparts (accounting for 2 cells).
2. **Missing egress baseline in N-F1:** because **N-F1** defaults egress to allow-all alongside ingress, it inherently provides the open connectivity required for **R-F3**×**N-F1** (reaching `X3`) and **R-F4**×**N-F1** (reaching `X5`) without requiring any additional **N-F2** weakness (2 cells).
3. **Policy overrides and ineffective controls:** ineffective policies (**N-F3**) or overly broad CNI/admin overrides (**N-F6**) reopen the exact connectivity left exposed by a missing baseline. Thus, every row that collapses under **N-F1** also collapses under **N-F3** and **N-F6**, specifically **R-F1** into `X1`, **R-F3** into `X3`, and **R-F4** into `X5` (6 cells).

Three of these cells (**R-F4**×{**N-F1**, **N-F3**, **N-F6**}) carry an additional complication: alongside `R-P4.4`, which drives the collapse into `X5`, they also contain an inert control-plane mechanism (`R-P4.5` or `R-P4.2`). Because RBAC composes with NetworkPolicy only when its action produces network-traversed traffic or directly targets network resources (for example `X8`/`X9`), control-plane-only operations remain independent. These instances are designated as CP in Table 7.5 and denoted alongside the primary collapse in Table 7.1 as COLL→`X5`/CP.

- **Borderline substrate** (eight cells, marked SUB): The foundation families **R-F5** and **N-F5** represent systemic preconditions rather than active attacker capabilities, forming no distinct joint capability when paired with standard families from the opposing domain.
- **Network-only capability** (two cells, marked NET): The exposure is driven entirely by the network plane, rendering the associated RBAC identity irrelevant. This independence was experimentally confirmed via the control experiment on **R-F1**×**N-F4** (Chapter 8).

Combined, these three mechanisms account for ten cells ($2 + 2 + 6$). Together with the eight cells of borderline substrate and the two cells of network-only capability, they fully account for all twenty non-composing cells in Table 7.1.

Table 7.7: The ten cells that collapse into an existing pattern instead of opening a new one, with the sub-pattern each side instantiates.

Cell	RBAC instance	NetPol instance	Into	Why
R-F1×**N-F3**	R-P1.3/R-P1.1	N-P3.1.1	X1	An ineffective policy that matches no pod is, for the attacker, no policy at all.
R-F1×**N-F6**	R-P1.3/R-P1.1	**N-F6** (family level)	X1	An over-broad cluster-wide or admin override reopens the connectivity **N-F1** lacks by default.
R-F3×**N-F6**	R-P3.1	**N-F6** (family level)	X3	Same substrate weakness as above, landing on X3's Secret-read mechanism instead.
R-F2×**N-F1**	R-P2.1.2	N-P1.1	X1	A single wildcard rule grants the same broad API reach as an overexposed identity.
R-F2×**N-F2**	R-P2.1.2	N-P2.2	X4	Same rule-level excess, paired with open egress instead of a flat network.
R-F3×**N-F1**	R-P3.1	N-P1.1	X3	A missing baseline defaults egress to allow-all too, the same channel N-P2.2 opens explicitly.
R-F3×**N-F3**	R-P3.1	**N-F3** (family level)	X3	An ineffective policy is indistinguishable from a missing baseline for the attacker, so it opens the same egress.
R-F4×**N-F1**	R-P4.4	N-P1.1	X5	A missing baseline defaults egress to allow-all too, the same channel N-P2.2 opens explicitly; R-P4.5 and R-P4.2 stay control-plane-only here regardless (X2, Chapter 8).
R-F4×**N-F3**	R-P4.4	**N-F3** (family level)	X5	An ineffective policy is indistinguishable from a missing baseline for the attacker, so it opens the same egress; the same control-plane-only instances (R-P4.5, R-P4.2) stay inert regardless.
R-F4×**N-F6**	R-P4.4	**N-F6** (family level)	X5	Same substrate weakness as above, landing on X5's pod-creation mechanism instead; R-P4.5 and R-P4.2 remain inert here too.

7.5 The five cross-families

The cross-taxonomy is organised by the capability the combination unlocks, which is a different axis from the one used in the two single domain chapters, for the reason given in Section 7.3. Each cross-family groups the patterns that reach the same capability and the last one, **XF5**, is a BORDERLINE foundation family that mirrors **R-F5** and **N-F5**. Every pattern is presented with the same schema, listed in Table 7.8.

Field	What it reports
Composition	the two weaknesses that combine
Equivalent patterns	alternative instances, on either side, that reach the same capability
Why this family, not another	disambiguates from a similar-looking pattern, where needed
Resulting capability	what the pair unlocks
Joint predicate	a single check over both objects that decides whether a cluster exhibits the pattern
Minimal fix	the single remediation that breaks the chain
Validation	what the lab in Chapter 8 found when it reproduced the pattern directly

Table 7.8: The schema used to present every cross-pattern in this chapter. Equivalent patterns and Why this family, not another appear only where needed.

Patterns without direct CVE evidence are analytically derived, as explained in Section 7.2.

7.5.1 XF1 — Lateral Movement

Effect. Reach and act on workloads beyond the entry point.

This family keeps a single pattern. It was proposed with two, but the lab in Chapter 8 refuted the second, `X2`, which paired `exec` with an unisolated peer. The action turned out to be served by the API server, so the network side was never necessary and `exec`'s own instance contributes nothing to that cell, which instead composes through pod creation into `X5` (Section 7.4). `X1` stays as the family's only member because lateral movement is a capability the matrix still has to account for.

MITRE ATT&CK mapping.

Technique	Why it applies
Use Alternate Authentication Material `T1550`	`X1`: the credential read from the peer's Secret is presented directly to the peer as authentication material, not as an account that belongs to the attacker

`X1` — Over-exposed identity on a flat network

Composition. **R-F1**/`R-P1.3` or `R-P1.1` × **N-F1**/`N-P1.1`.

→ `R-P1.3`/`R-P1.1`: a workload's ServiceAccount bound to a role with a cluster-takeover capability. The lab instantiates `R-P1.3.1`, Admin Overprivilege, a wildcard verb over Secrets, cluster-wide, rather than a narrow read tied to one Secret.

→ `N-P1.1`: no default-deny baseline, so pods reach any peer in the cluster.

Equivalent patterns: the ClusterRole bound to the victim's ServiceAccount grants a wildcard verb over Secrets, cluster-wide (Section 8.2) and the lab instantiates `R-P1.3`'s own cluster-takeover grant, `* secrets`.

It is not the only instance that reaches `X1`: in fact, Table 7.7 shows a stricter **R-F2** rule such as `R-P2.1.2` getting there by a different route, so the choice here is which instance the lab runs, not the only one available.

On the network side, an over-broad **N-F2** rule that leaves `from`/`to` empty (`N-P2.1.1`, Section 6.3) opens the same unrestricted reachability as `N-P1.1`, and so does an ineffective policy (**N-F3**, collapsed here per Section 7.4). Because `N-P1.1.1`, the dominant member of **N-F1**, leaves both ingress and egress open at once (Section 6.2.1), the same pod already satisfies `X4`'s egress precondition too, without a separate **N-F2** weakness.

Why this family, not another. One might expect **R-F3**×**N-F1** to collapse into `X1` rather than `X3`, since `N-P1.1` is the same network instance `X1` itself uses. It does not, because the two RBAC instances grant different things. `R-P1.3` and `R-P2.1.2` grant broad reach over Kubernetes resources, the foothold `X1` needs to act on other workloads. `R-P3.1` grants a read on Secret content, credentials to exfiltrate by default. It only reaches `X1`'s mechanism when the secret read happens to authenticate to a specific reachable peer, an instance-level overlap the family-level granularity does not resolve (Section 7.6). A missing baseline opens the same egress regardless of which RBAC instance sits behind it, so what decides the outcome is the RBAC side, not the network side, and `R-P3.1` lands on `X3` at the family level (Table 7.7).

Resulting capability. The broad token reads a credential that belongs to the peer and the flat network opens a connection to that peer, where the credential authenticates the attacker as a legitimate client. The token alone has nothing to deliver it to. The network alone has nothing to present.

JOINT PREDICATE

$\exists\, sa :$ `broadAPI`(sa) $\wedge$ `reachable`(attackerPod, peer)

Minimal fix. Scope down the ServiceAccount on the RBAC side, or apply a namespace default-deny on the network side. The default-deny is the cheaper choice.

Validation. A dedicated lab (Section 8.2) tested both single fixes directly. Scoping the ServiceAccount refuses the read outright. Applying the default-deny still lets the read succeed but leaves the peer unreachable, confirming that either half alone breaks the chain.

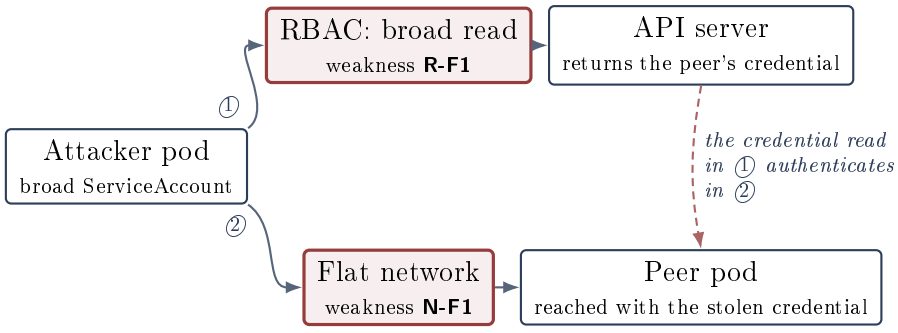


Figure 7.1: How X1 chains the two weaknesses: the broad ServiceAccount lets the attacker read the peer's credential through the API (**R-F1**) and the flat network carries that credential to the peer, where it authenticates the attacker as a legitimate client (**N-F1**).

7.5.2 XF2 — Data Exfiltration

Effect. Move sensitive data out of the cluster.

MITRE ATT&CK mapping.

Technique	Why it applies
Unsecured Credentials T1552	X3: the Secret read exposes credentials that open egress then carries out of the cluster
Unsecured Credentials T1552	X4: open egress carries the mounted ServiceAccount token out of the cluster
Valid Accounts T1078	X4: the exfiltrated token is then reused from outside as a valid account
Deploy Container T1610	X5: the identity deploys a purpose-built pod that then egresses freely
Exfiltration Over C2 Channel T1041	X3/X4/X5: the open egress path is what carries the credential or the data out of the cluster

X3 — Secret read with unrestricted egress

Composition. **R-F3**/R-P3.1 × **N-F2**/N-P2.2.

→ R-P3.1: the identity can read Secrets.

→ N-P2.2: egress to 0.0.0.0/0.

Equivalent patterns: **N-F2** is not the only way to open the egress X3 needs. Network side, N-P1.1.1, the dominant member of **N-F1**, leaves both ingress and egress open at once (Section 6.2.1), so a missing baseline already satisfies the same egressAllowed precondition N-P2.2 states explicitly or an other case is an ineffective policy (**N-F3**) which is indistinguishable from a missing baseline for the attacker

and collapses the same way (Section 7.4). On the RBAC side, **R-F1**'s `R-P1.3` and **R-F2**'s `R-P2.1.2` are broad enough to read Secrets too, but their excess reaches across the whole API rather than Secrets specifically, so those cells land on `X4` instead: exfiltrating the identity's own broad token, not a Secret it has no legitimate reason to hold (Table 7.7).

Resulting capability. Read the secrets and then send them to an endpoint the attacker controls. The read achieves nothing without a way out and open egress is harmless with nothing sensitive to read.

JOINT PREDICATE

$\texttt{canReadSecrets}(id) \wedge \texttt{egressAllowed}(pod, \texttt{0.0.0.0/0})$

Minimal fix. Restrict egress to known destinations on the network side, which also contains several other exfiltration patterns, or remove the read on Secrets.

Validation. Table 8.1 reports the same pattern here. Removing the read on Secrets blocks it outright. Restricting egress still lets the read succeed while the send itself is blocked.

`X4` — ServiceAccount token exfiltration

Composition. **R-F1**/`R-P1.3` × **N-F2**/`N-P2.2`.

→ `R-P1.3`: a broad identity. Kubernetes mounts every ServiceAccount's token into its pod by default, regardless of RBAC. What RBAC contributes here is not the mount but the breadth of what that mounted token authorizes once exfiltrated, which is what makes it worth stealing.

→ `N-P2.2`: open egress.

Why this family, not another. `X4` could look like `X1`, since both use **R-F1**/`R-P1.3`, or like `X3`, since both exfiltrate a credential over open egress. It is neither: the token used in `X4` is not obtained through an explicit RBAC action in the way `X1`'s peer credential or `X3`'s Secret is; Kubernetes mounts it into the pod independently of RBAC. What `R-P1.3`'s broad grant contributes is not access to a new resource, but the breadth of what that mounted token is authorised to do once carried outside the cluster. That breadth is why the excess sits in **R-F1** rather than **R-F3**: **R-F3**'s `R-P3.1` reads a Secret that the identity has no legitimate reason to access, whereas `X4` concerns the excessive authority of a credential already available inside the pod. `X4` also differs from `X1` in whose credential moves and where it goes. `X1` carries a credential belonging to another workload back to that same workload, to authenticate against it from inside the cluster. `X4` carries the attacker's own credential outside the cluster, where it can be reused from an external location. The same missing baseline can open both paths at once, which is exactly the overlap accounted for by the equivalent patterns below.

Equivalent patterns. **R-F1**'s own instance is not the only way to reach `X4`. A single over-reaching **R-F2** rule, such as `R-P2.1.2`, can grant the same broad API authority to the identity represented by the mounted token, reaching `X4` through a permission-overreach condition rather than an overexposed identity (Table 7.7).

On the network side, N-P1.1.1, the dominant member of **N-F1**, already leaves egress open together with ingress (Section 6.2.1), so a missing baseline satisfies X4's precondition without requiring a separate **N-F2** weakness (Section 7.5.1).

Resulting capability. Send the mounted token out of the cluster and reuse it from outside, which separates the stolen identity from the compromised pod.

JOINT PREDICATE

broadAPI(sa) $\wedge$ egressAllowed(pod, external)

Minimal fix. Restrict egress on the network side, or stop mounting the token where it is not needed.

Validation. The lab confirms both halves (Table 8.1). Scoping the identity leaves the token useless, a 403 on every call. Restricting egress leaves the token valid but keeps it from ever leaving the cluster.

X5 — Pod creation egress channel

Composition. **R-F4**/R-P4.4 $\times$ **N-F2**/N-P2.2.

$\rightarrow$ R-P4.4: the identity can create pods.

$\rightarrow$ N-P2.2: the created pod is placed in a context where egress is permitted.

Equivalent patterns. **N-F2** is not the only way to open the egress X5 needs. N-P1.1.1, the dominant member of **N-F1**, leaves both ingress and egress open at once (Section 6.2.1), so a missing baseline already satisfies the same egressAllowed precondition N-P2.2 states explicitly. An ineffective policy (**N-F3**) is indistinguishable from a missing baseline for the attacker and collapses the same way, and so does an over-broad cluster-wide or admin override (**N-F6**), which reopens the same connectivity **N-F1** lacks by default (Table 7.7).

Resulting capability. Create a pod built for the purpose and let it egress freely to send data out. The escalation verb supplies the workload and the open egress supplies the channel.

JOINT PREDICATE

canCreatePods(id) $\wedge$ egressAllowed(ns, external)

Minimal fix. A default-deny egress baseline at namespace scope, or the removal of pod creation on the RBAC side.

Validation. Both fixes were tested directly (Table 8.1). Removing pod creation refuses it outright, a 403 on the first step. Blocking egress still lets the pod be created but denies it any path out.

7.5.3 XF3 — Cross-Boundary Access

Effect. Reach a sensitive endpoint that network policy should have fenced off.

MITRE ATT&CK mapping.

Technique	Why it applies
Cloud Instance Metadata API T1552.005	X6: the pod queries the cloud metadata API and takes the node's credentials
Deploy Container T1610	X6: pod creation is what deploys the container that reaches the metadata endpoint
Container and Resource Discovery T1613	X7: reading ConfigMaps or endpoint objects discovers an internal service the network then lets the actor reach

X6 — Pod creation reaching cloud metadata

Composition. R-F4/R-P4.4 × N-F4/N-P4.1.

→ R-P4.4: the identity can create pods.

→ N-P4.1: the metadata service at 169.254.169.254 is reachable.

A related route through R-P4.6 (substrate access, such as the kubelet API proxy) can reach the same node credentials without creating a pod at all. It is a distinct instance with its own reachability condition, not covered by the joint predicate below, which is scoped to the pod-creation route.

Equivalent patterns. N-F4 is not the only way to reach the metadata endpoint X6 needs. N-P4.1's own detection criterion (Section 6.5.1) is satisfied by the absence of egress isolation as well as by an explicit rule, so a missing baseline (N-F1) or an ineffective policy indistinguishable from one (N-F3) already reaches the same endpoint, without a separate N-F4 weakness. On the RBAC side, R-P4.4 is not the only plausible instance either: R-F1's broad identity or a R-F2 over-reaching rule could analytically carry enough create access to the same pod-creation route, though no entry in the corpus documents it directly.

Resulting capability. Schedule a pod that queries the metadata API. Where the endpoint is exposed, this may obtain the cloud credentials of the node and turn a foothold inside the cluster into a compromise of the cloud account, though the exact exposure still depends on the cloud provider and the IMDS configuration in place.

JOINT PREDICATE

$\texttt{canCreatePods}(id) \wedge \texttt{reachable}(pod, \texttt{169.254.169.254})$

Minimal fix. Block egress towards the metadata address on the network side, which is the single most effective choice, or remove pod creation on the RBAC side.

Validation. The same two fixes were tested here (Table 8.1). Removing pod creation refuses it outright, the same 403 as X5. Blocking egress to the metadata address still lets the pod be created but the endpoint becomes unreachable.

X7 — Endpoint info disclosure meets reachability

Composition. R-F3/R-P3.2 × N-F4/N-P4.1.

→ R-P3.2: the identity can read ConfigMaps or endpoint objects that reveal internal addresses.

→ N-P4.1: those endpoints are reachable.

Equivalent patterns. N-F4 is not the only way to reach the endpoint X7 needs. N-P4.1's own detection criterion (Section 6.5.1) is satisfied by the absence of egress isolation as well as by an explicit rule, so a missing baseline (N-F1) or an ineffective policy indistinguishable from one (N-F3) already reaches the same endpoint, without a separate N-F4 weakness. On the RBAC side, R-P3.2 is not the only plausible instance either: R-F1's broad identity or a R-F2 over-reaching rule could analytically carry enough read access to the same ConfigMap or endpoint objects.

Resulting capability. The read locates a sensitive internal service such as an admin API, a database or a dashboard and the open path lets the actor connect to it. Knowledge and reachability together amount to access.

JOINT PREDICATE

$\texttt{canReadEndpointInfo}(id) \wedge \texttt{reachable}(pod, \text{sensitiveEndpoint})$

Minimal fix. Isolate the sensitive endpoint on the network side, or restrict the read on configuration and endpoint objects.

Validation. Table 8.1 confirms both halves. Restricting the read on ConfigMaps and endpoint objects blocks discovery outright. Isolating the sensitive endpoint still lets the address be read but the endpoint itself becomes unreachable.

7.5.4 XF4 — Defense Evasion via the Network Control Surface

Effect. Disable a network isolation control that was written correctly, by acting on it directly through RBAC.

This family holds the two disabling patterns, where an RBAC permission acts directly on the objects that implement network isolation. It is the clearest example of the two domains interacting and it is almost absent from both single domain datasets.

MITRE ATT&CK mapping.

Technique	Why it applies
Disable or Modify Tools T1685	X8: the permission edits or deletes the NetworkPolicy objects that enforce isolation
Disable or Modify Tools T1685	X9: the dual-axis wildcard grant lets the actor rewrite the CNI's own policy CRDs, disabling enforcement platform wide

X8 — Disabling isolation by modifying or deleting NetworkPolicy objects

Composition. R-F2/R-P2.2.3 ⇒ N-F3/N-P3.1.1.

→ R-P2.2.3: manage-all over the networking.k8s.io API group.

→ N-P3.1.1: the isolating policy is retargeted to a selector that matches nothing and becomes inert.

Equivalent patterns. R-P2.2.3 is not the only instance that disables isolation this way. Because the target's apiGroup, networking.k8s.io, is fixed and known, a wildcard on either axis alone (R-P2.1.2 or R-P2.1.3, Section 5.3) already reaches it, a broader grant that happens to cover this resource too (Table 7.3). This is weaker than what X9 needs: there the CNI's apiGroup is unknown, so only a wildcard on both axes at once rules it in. On the network side, retargeting the selector (N-P3.1.1) is not the only way the write access disables isolation either. Deleting the NetworkPolicy outright reaches the same result through N-F1's own missing-baseline state instead, rather than a second N-F3 sub-pattern.

Resulting capability. The attacker does not need to find a flat network, because one can be made by disabling the policies that enforce isolation, either by deleting them or by retargeting their selector so that they govern nothing. This reaches the same state N-P3.1.1 detects as an authoring mistake, produced here instead by the attacker's own write access. X1 follows from there. The permission belongs to the control plane and its object is the control of the data plane.

JOINT PREDICATE

canWrite(id, networkpolicies) ⇒ canDisable(isolation)

Minimal fix. On the RBAC side only. Write and delete on networkpolicies should never be granted outside a tightly controlled platform role and bindings that carry it should be audited explicitly.

Validation. The disabling design tests only the RBAC side (Table 8.1), since there is no independent network weakness to remove separately. Restricting write and delete on networkpolicies refuses the edit outright, the same 403 that would stop the disabling action itself.

X9 — Wildcard permissions over CNI and admin policy objects

Composition. R-F2/R-P2.1.2 and R-P2.1.3 ⇒ N-F6 (Appendix A).

→ R-P2.1.2/R-P2.1.3: the rule wildcards resources and apiGroups with a write verb. A wildcard on only one axis is not enough, since a fixed resource name or apiGroup still rules out the CNI's own CRDs.

→ N-F6: enforcement relies on the policy CRDs of the CNI or on AdminNetworkPolicy. The network side is named at family level, since N-F6 has no subpattern for a policy that has been subverted.

Equivalent patterns. `R-P2.1.2` and `R-P2.1.3` together are not the only instance. A cluster-wide manage-all grant (`R-P2.2.1`, Section 5.3) already wildcards every resource and apiGroup at once, so it reaches the CNI's own CRDs the same way, without needing the two narrower wildcards to line up on the same rule.

Resulting capability. A rule that wildcards `resources` and `apiGroups` together silently includes authority over the policy CRDs of the CNI in use, whichever CNI that is, since neither axis names them and no combination of the two rules them out. An actor holding it can rewrite the cluster-wide admin policies that override the namespaced ones. This links a dual-axis over-reach of Chapter 5 to the one NetworkPolicy family that existing tooling does not cover.

JOINT PREDICATE

`wildcard`(*rule*.resources) ∧ `wildcard`(*rule*.apiGroups) ∧ `canWrite`(*rule*)

Minimal fix. Remove the wildcard from either `resources` or `apiGroups` in Roles and ClusterRoles and enumerate it explicitly, so that the policy CRDs of the CNI are never granted by accident.

Validation. The same disabling design applies here (Table 8.1). Removing either wildcard refuses the policy delete outright, a 403 on the call that would otherwise disable enforcement platform wide.

7.5.5 XF5 — Authorizer Disabled and Substrate Not Enforcing

Effect. None in the usual sense, since the family describes a substrate precondition rather than an attacker action.

This family mirrors **R-F5** and **N-F5**. Neither plane enforces, so what it composes is two absent guardrails rather than two capabilities. It is borderline for exactly this reason.

MITRE ATT&CK mapping. `X10` is a precondition that amplifies every other tactic rather than mapping onto one of its own, so no single technique is listed here.

`X10` — Authorizer disabled and substrate not enforcing

Composition. **R-F5**/`R-P5.1` × **N-F5**/`N-P5.1` or **N-F3**/`N-P3.1`.

→ `R-P5.1`: the RBAC authorizer is disabled.

→ `N-P5.1`/`N-P3.1`: the CNI does not enforce NetworkPolicy, or the policies are inert.

Resulting capability. The cluster authorises every API call and permits every connection, so both planes are open at the same time. The impact reaches the whole cluster and the preconditions of several other cross-patterns are trivially satisfied under it.

JOINT PREDICATE

¬ `rbacEnforced`(apiserver) ∧ ¬ `netpolEnforced`(CNI)

Minimal fix. There is none in the usual sense, because both sides are foundational. The `-authorization-mode` has to include RBAC and the cluster has to run a CNI that enforces NetworkPolicy. Either fix restores one plane and both are needed.

Validation. `X10` was not reproduced in the lab (Chapter 8). It describes a substrate precondition rather than an attacker action, so there is no single call whose refusal would demonstrate it.

7.6 Scope, limits and summary

Table 7.9 consolidates the cross-taxonomy: five cross-families, eight cross-patterns and one borderline foundation condition (`X10`), spanning every composing cell of Table 7.1.

Scope. The work covers RBAC and NetworkPolicy misconfigurations and their composition. Other control planes, PodSecurity and admission in particular, with privileged pods, `hostPath` and `hostNetwork`, are deliberately out of scope.

Declared limits.

- **Families, not patterns.** The matrix works at the level of families, while each cross-pattern names the specific pattern it builds on, because within one family not all patterns compose in the same way. `X5` and `X6` both build on `R-P4.4` and differ only in the network weakness they meet.
- **Pairs only.** Real attack chains are longer than a pair and are obtained by putting cross-patterns one after another, as in `X8` → `X1` → `X6`. `X8` first disables the NetworkPolicies that isolate the cluster, either by deleting them or by retargeting their selector so they match nothing, which leaves the network without working isolation either way (Table 7.7). That satisfies the network side `X1` needs, so an attacker whose own ServiceAccount already carries the broad reach of **R-F1** uses it to read a peer's credential and reach that peer directly, exactly as Figure 7.1 shows. If the peer, or the identity the attacker now controls, can also create pods (`R-P4.4`), the same missing isolation that carried `X1` reaches the cloud metadata endpoint too, without a separate **N-F4** weakness, exactly as `X6`'s own equivalent patterns allow, and a pod scheduled against it turns the foothold into a compromise of the cloud account. The taxonomy still records each step as an independent cross-pattern; chaining them into sequences like this one is future work.
- **No measurement on real clusters.** The patterns are derived analytically, since the tooling is single domain and only two CVEs span both domains. How often each one occurs in practice is not measured. Beyond the CVE now cited for `R-P4.4`, the remaining patterns still rely on the lab validation of Chapter 8

rather than a documented case in the wild, so frequency in practice stays future work.

Table 7.9: The cross-taxonomy at a glance: five capability-based families, eight cross-patterns and one borderline foundation condition, each as RBAC family × NetworkPolicy family.

Cross-family	Pattern	Composition (RBAC × NetPol)
XF1 Lateral Movement	X1	**R-F1** broad identity × **N-F1** missing isolation
XF2 Data Exfiltration	X3	**R-F3** read Secrets × **N-F2** open egress
	X4	**R-F1** broad token × **N-F2** open egress
	X5	**R-F4** create pods × **N-F2** open egress
XF3 Cross-Boundary Access	X6	**R-F4** create pods × **N-F4** metadata reachable
	X7	**R-F3** read endpoint info × **N-F4** endpoint reachable
XF4 Defense Evasion (disabling)	X8	**R-F2** write `networkpolicies` ⇒ **N-F3** inert
	X9	**R-F2** resources+apiGroups wildcard ⇒ **N-F6** CNI and admin policy CRDs
XF5 Authorizer Disabled, Not Enforcing (BL)	X10	**R-F5** authz disabled × **N-F5** substrate not enforcing

The numbering runs from X1 to X10 and X2 is missing. That code belonged to a pattern which satisfied the composition criterion on paper and was written up as one, until the validation of Chapter 8 (Section 8.3) showed that its network side was not necessary and the pattern was withdrawn. The code is left unused so that the two chapters can be read against each other. This is the only instance where the experiment overturned the analysis, rather than confirming an exclusion the analysis had already made.

Chapter 8

Practical validation of the cross-domain patterns

As in the two previous chapters, the two single domain taxonomies derive their patterns bottom up, from tooling and CVEs, while the cross-domain layer is not covered by any tool surveyed online, so its patterns had to be researched instead.

If two weaknesses really do combine into something neither produces alone, that should be visible in a running cluster and this chapter checks whether it is. For each pattern it reproduces the attack and tests the joint necessity criterion of Section 7.3 against what the cluster actually does.

8.1 Method

The validation uses a reproducible lab built on `kind` with the Calico CNI because it enforces policies while the default `kindnet` does not and with it every network side fix would pass without doing anything. Each case runs in its own namespace on a shared cluster, in one of three designs.

Compositions (×). A pattern that composes two independent weaknesses is tested with a three configuration comparison:

- **A — complete**: both weaknesses present so the attack is expected to succeed.
- **B — RBAC fixed**: only the RBAC weakness removed, so it is expected to fail.
- **C — network fixed**: only the NetworkPolicy weakness removed, so it is expected to fail.

If A succeeds while B and C fail, the capability exists only when both weaknesses hold. A differs from B by the RBAC side alone and from C by the network side alone, so each failure points at the plane that was removed.

Disabling patterns (⇒). For the two disabling patterns (`X8`, `X9`) there are not two independent weaknesses but a single RBAC permission that acts on a network object. The test starts from a precondition where the isolation is in place and the target is unreachable. Case A then gives the attacker the RBAC permission and

checks whether the isolation is removed and the target reached and case B removes the permission again and checks that the isolation holds. The RBAC side alone decides the outcome here, since there is no network weakness to remove.

Excluded cells. The same comparison, with the outcome reversed, tests cells that are not crosses. If fixing one side leaves the attack working, that side is not necessary and the capability is single domain.

Figure 8.1 summarises the three designs and what each expects to see.

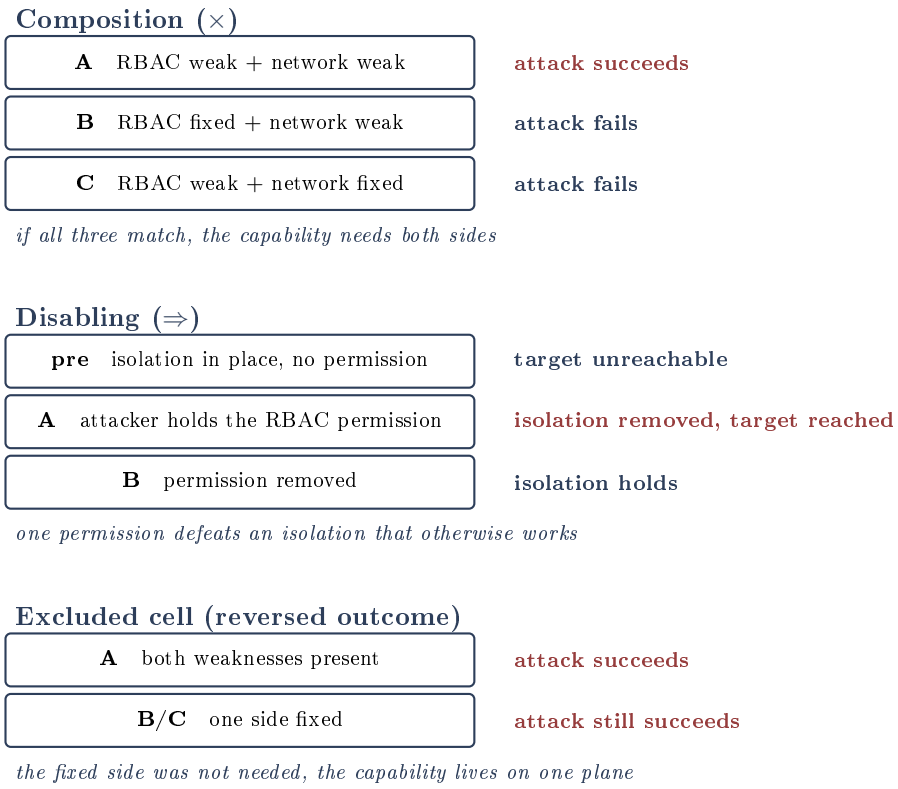
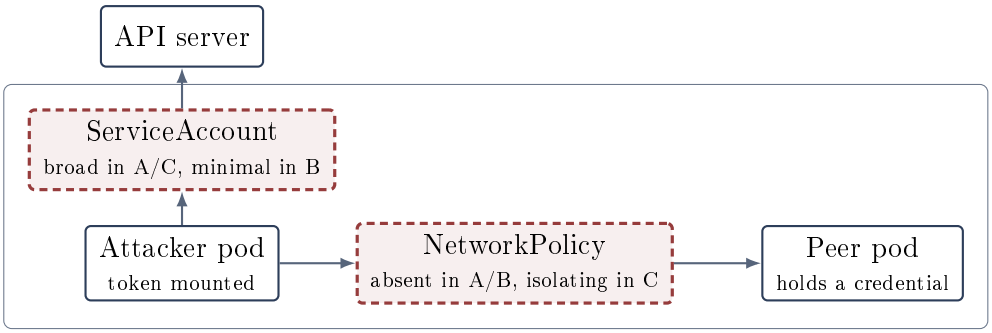


Figure 8.1: A composition is confirmed when removing either weakness stops the attack and a disabling pattern is confirmed when one permission is enough to undo an isolation that otherwise holds. An excluded cell is confirmed when fixing one side changes nothing.

Testing precondition. Every privileged action is performed from inside the compromised pod, using the ServiceAccount token mounted in it. This is not just a convenience. It is how the tests have to be done. Running `kubectl` with an explicit `--token` from an administrator's kubeconfig does not authenticate as that token, since the client certificate in the kubeconfig wins the TLS handshake and the request is authorised as the administrator. Inside the pod, the ServiceAccount is the identity the API server actually evaluates.



one namespace, one lab case (`kind` cluster, Calico CNI)
only the ServiceAccount binding and the NetworkPolicy change between configurations A, B and C, the rest of the namespace stays fixed

Figure 8.2: The Kubernetes objects behind the three-configuration comparison of Figure 8.1, using `X1`'s namespace as the running example, where only the RoleBinding and the NetworkPolicy are toggled between cases A, B and C.

8.2 Two cases in full

The eight labs follow two designs, composition and disabling, and one case from each is reported here end to end while the remaining six are summarised in Table 8.1. `X1` is the clearest composition, because its two failures have visibly different causes, and `X8` is the simpler of the two disabling patterns. Repeating either walkthrough six more times would add length without adding evidence, since Table 8.1 already prints the exact outcome each configuration produced rather than a restated conclusion.

Composition — `X1` — Over-exposed identity on a flat network

Section 7.5 declares this composition as `R-P1.3`, Privileged SA distribution (family **R-F1**), joined with `N-P1.1`, Missing default-deny baseline (family **N-F1**). The lab builds a namespace with a peer holding a credential, a victim pod and a ServiceAccount bound to a ClusterRole that grants exactly the `R-P1.3.1` weakness, a wildcard verb over Secrets, cluster-wide, rather than a narrow read tied to one Secret. The attack itself only ever calls `get`, but the identity is broad enough to read, create or delete any Secret in the cluster, which is the cluster-takeover capability the pattern's detection criterion asks for. The attack always runs from inside the victim, with the token mounted in it. This is the ClusterRole:

✗ **Vulnerable manifest** (triggers R-P1.3.1)

```
# A wildcard verb on secrets, cluster-wide, is Admin Overprivilege
# (R-P1.3.1), not merely a read. The attack only ever calls get, but the
# ServiceAccount could read, create or delete any Secret in the cluster.
apiVersion: v1
kind: ServiceAccount
metadata:
  name: app-sa
  namespace: x1-demo
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
```

```
  name: secrets-admin-wide-x1
rules:
  - apiGroups: [""]
    resources: ["secrets"]
    verbs: ["*"]                      # <-- triggers R-P1.3.1
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: app-sa-secrets-wide-x1
subjects:
  - kind: ServiceAccount
    name: app-sa
    namespace: x1-demo
roleRef:
  kind: ClusterRole
  name: secrets-admin-wide-x1
  apiGroup: rbac.authorization.k8s.io
```

The attack follows two steps:

1. read the peer's credential through the API;
2. use that credential against the peer over the network.

These are the two steps as `attack.sh` actually runs them, from inside the victim pod, with the token mounted in it:

Lab script — attack.sh, core exploit

```
# Read the peer's credential through the API, using the token mounted
# in the ServiceAccount above (step one).
T=$(cat /var/run/secrets/kubernetes.io/serviceaccount/token)
CA=/var/run/secrets/kubernetes.io/serviceaccount/ca.crt
API=https://kubernetes.default.svc
RESP=$(curl -sS -m 10 --retry 5 --retry-delay 1 --retry-connrefused --cacert "
    $CA" \
        -H "Authorization: Bearer $T" "$API/api/v1/namespaces/x1-demo/secrets/
            peer-cred")
PW=$(echo "$RESP" | jq -r ".data.password // empty" 2>/dev/null | base64 -d 2>/
    dev/null)

# Use the stolen credential against the peer over the flat network (step two).
curl -s -m 8 "$PEER/?pw=$PW" 2>/dev/null | grep -q "X1-PEER-OWNED"
```

The three cases toggle what the victim's binding and the peer's NetworkPolicy allow:

1. **Case A** leaves both weaknesses in place.
2. **Case B** replaces the binding with a minimal one.
3. **Case C** leaves the binding alone and isolates the peer with a NetworkPolicy.

`N-P1.1` is the absence of any such policy in the namespace. The fix applied only in Case C supplies exactly the baseline that was missing:

Lab manifest — Case C isolation

```
# Applied only in Case C. It denies all ingress to the peer, the N-P1.1
# baseline the namespace lacked. The read in step one still succeeds, but the
# stolen credential now has nowhere to reach.
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: isolate-peer
  namespace: x1-demo
```

```
spec:
  podSelector:
    matchLabels: { app: peer }
  policyTypes: ["Ingress"]
  ingress: []
```

Lab output — X1

```
CASE A  complete       (expected: SUCCEEDS)
  -> READ-OK: peer credential stolen
  [RESULT] SUCCEEDED -> credential stolen AND used against a reachable peer.

CASE B  RBAC fixed     (expected: FAILS)
  -> READ-FAILED (RBAC denies reading the Secret)
  [RESULT] FAILED -> one side missing.

CASE C  network fixed  (expected: FAILS)
  -> READ-OK: peer credential stolen
  -> REUSE-FAILED (NetworkPolicy denies reaching the peer)
  [RESULT] FAILED -> one side missing.
```

The two failures are not the same failure. In case B the read itself is refused, so the attack stops before the network is ever used, while in case C the read succeeds, the attacker holds the credential and the attempt dies on the way to a peer that is no longer reachable. This is what the joint necessity criterion of Section 7.1 asks for and it says more than the observation that both fixed cases fail, because each fix breaks a different step, which is only possible if the two weaknesses were contributing different things.

Disabling pattern — X8 — Disabling isolation by editing NetworkPolicy objects

Section 7.5 declares this disabling relation as R-P2.2.3, Single-API-group manage-all (family **R-F2**), acting on N-P3.1.1, Dangling policy (family **N-F3**). Unlike a composition, there is a single permission to test rather than two independent weaknesses. The RBAC side does not merely coexist with the network defect, it is what creates it. The lab builds a namespace with a target service, an isolating NetworkPolicy and a victim pod whose ServiceAccount has manage-all over the entire `networking.k8s.io` API group, exactly the R-P2.2.3 weakness. It acts in one move, retargeting the NetworkPolicy's selector until it matches nothing. That is N-P3.1.1, a policy left in place but pointed at nothing, leaving the target unprotected without deleting the policy object.

✗ Vulnerable manifest (triggers R-P2.2.3)

```
# Full control (verbs=*, resources=*) over the entire networking.k8s.io API
# group. The ServiceAccount can create, modify or delete any NetworkPolicy.
apiVersion: v1
kind: ServiceAccount
metadata:
  name: app-sa
  namespace: x8-demo
---
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: networking-admin
  namespace: x8-demo
rules:
  - apiGroups: ["networking.k8s.io"]
```

```
    resources: ["*"]
    verbs: ["*"]                    # <-- triggers R-P2.2.3
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: app-sa-networking-admin
  namespace: x8-demo
subjects:
  - kind: ServiceAccount
    name: app-sa
    namespace: x8-demo
roleRef:
  kind: Role
  name: networking-admin
  apiGroup: rbac.authorization.k8s.io
```

This is the retarget, run from inside the victim pod with the token mounted in it, a single PATCH that only the permission above allows.

Lab script — attack.sh, core exploit

```
# Retarget the podSelector to a label no pod carries. The policy object
# stays untouched but now governs zero pods, which is R-P3.1.1, a dangling
# policy produced without ever deleting or disabling anything.
PATCH='{"spec":{"podSelector":{"matchLabels":{"app":"ghost-nonexistent"}}}}'
CODE=$(sa_api_patch "$NS" victim \
       "apis/networking.k8s.io/v1/namespaces/$NS/networkpolicies/$POLICY" "
           $PATCH")
```

Before it is retargeted, the policy is a correctly enforcing baseline, not the `N-P3.1.1` defect itself. The manifest below is the control the RBAC permission is used to defeat:

Lab manifest — network precondition

```
# Default-deny ingress to the target, the control the RBAC permission is
# used to defeat. This is the safe starting point the precondition below
# checks for, not a fix applied afterwards, nor the N-P3.1.1 defect itself.
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: isolate-target
  namespace: x8-demo
spec:
  podSelector:
    matchLabels: { app: target }
  policyTypes: ["Ingress"]
  ingress: []               # no ingress rule = deny all ingress to the target
```

The design has a precondition check plus two cases:

1. **Precondition.** With the NetworkPolicy in place, the target must be unreachable.
2. **Case A** retargets the selector with the RBAC permission in place, then tries to reach the target.
3. **Case B** repeats the same attempt once the permission on `networkpolicies` is removed.

Lab output — x8

```
PRECONDITION            (expected: target BLOCKED)
  -> OK: target blocked, there is a network control to break.

CASE A  disable         (expected: SUCCEEDS)
```

```
  -> PATCH-OK: NetworkPolicy selector retargeted to dangling (HTTP 200)
  [RESULT] TARGET REACHED -> isolation defeated via RBAC.

CASE B  RBAC fixed      (expected: FAILS)
  -> PATCH-FAILED: RBAC denies the retarget (HTTP 403)
  [RESULT] TARGET BLOCKED -> isolation still active.
```

Here a single fix is enough, unlike `X1`. Case A shows the permission alone defeats an isolation the precondition already confirmed was working, and Case B shows removing that one permission restores it completely. One side is weak and the other was already correct, which is exactly what marks `X8` as a disabling pattern rather than a composition.

8.3 Results

Table 8.1 summarises the outcome in which eight patterns behave as the taxonomy predicts. For the six compositions A succeeds and each single side fix breaks the chain and the last column records which signal was observed, so that the two fixes can be told apart. For the two disabling patterns the RBAC permission is enough to defeat an isolation that the precondition check showed was working and removing the permission leaves the target unreachable.

The pattern the lab refuted

One of the proposed patterns did not survive. `X2` paired `exec` or `attach` on a peer with the absence of isolation and on paper it satisfied the composition criterion, because the permission on its own reaches nothing if the peer cannot be reached. In the lab the peer was isolated with a NetworkPolicy in case C and the `exec` went through anyway.

Lab output — X2

```
CASE A  complete        (expected: SUCCEEDS)
  -> EXEC-OK: attacker exec'd into the peer, marker read.
  [RESULT] SUCCEEDED -> as expected.

CASE B  RBAC fixed      (expected: FAILS)
  -> EXEC-FAILED: pods/exec denied by RBAC.
  [RESULT] FAILED -> as expected.

CASE C  network fixed   (expected: FAILS)
  -> EXEC-OK: attacker exec'd into the peer, marker read.
  [RESULT] SUCCEEDED -> NOT as expected: the isolated peer was still reached.
```

The reason is structural rather than accidental, since a request to `pods/exec` is served by the API server, which opens the session towards the kubelet of the node where the target runs, so the traffic never travels between the two pods and the policy plane never sees it. No NetworkPolicy can influence something it does not carry, which means the network side of the pair was never necessary and the capability belongs to a single domain, so `X2` was withdrawn: `exec`'s own instance contributes nothing to that cell, which instead composes through pod creation into `X5` (Section 7.4).

The same reasoning covers more than one cell, because it holds for every subresource that the API server serves instead of the pod network carrying it, `attach`

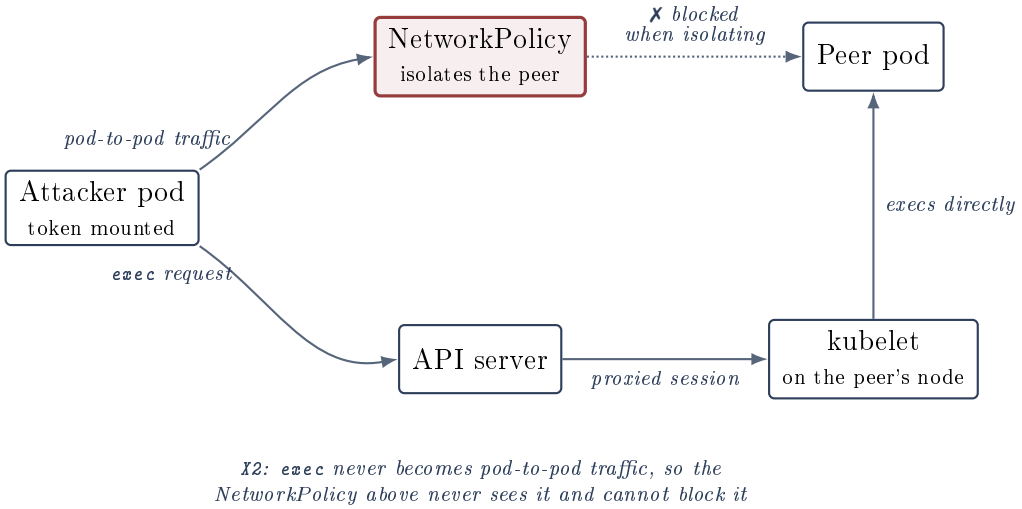


Figure 8.3: The two paths to the peer pod: ordinary traffic over the pod network, which NetworkPolicy inspects, and the `kubectl exec` path through the API server and kubelet, which does not.

and `portforward` included and it is also the one result the analysis alone could not have reached, which is why building the labs mattered.

Experimental Validation and Refutation Analysis

The control experiments, the reversed tests, confirmed the theoretical expectations about the independence of specific domain conditions. Three cases required this kind of test, each resolved differently:

- **Confirmed exclusion: R-F1×N-F4.** This pairing asks whether obtaining the node's cloud credentials from the metadata service requires a broad identity (**R-F1**) on top of the network path to it (**N-F4**), or whether the network path alone already accounts for that capability. The service itself answers without authentication to whatever process reaches it (Section 6.5), so the request never passes through RBAC in the first place. The reversed test confirmed this empirically: it reduced the ServiceAccount to a minimal identity while leaving the network side unchanged, and the credentials remained retrievable regardless of the RBAC permissions. **R-F1×N-F4** therefore does not form a cross-domain composition and the capability, obtaining the node's cloud credentials, is fully explained by the network side alone. The RBAC side still matters for this same endpoint elsewhere as `X6` and `X7` reach it too, but through `R-P4.4` or `R-P3.2` respectively, patterns that place or point the attacker at the endpoint in the first place.

- **Reclassification: R-F3×N-F1.** This pairing asks whether combining Secret read access (**R-F3**) with a missing network baseline (**N-F1**) forms an exfiltration capability of its own, comparable to `X3`, or whether the read alone already accounts for obtaining the secret's contents regardless of the network state. The experiment required a more complex interpretation, checking this by applying a namespace default-deny policy in case C.

  Lab output — R-F3×N-F1

```
CASE A  complete        (expected: SUCCEEDS)
  -> READ-OK: Secret exfiltrated via the API server.
  [RESULT] SUCCEEDED -> as expected.

CASE B  RBAC fixed      (expected: FAILS)
  -> READ-FAILED: RBAC denies reading the Secret.
  [RESULT] FAILED -> as expected.

CASE C  network fixed   (expected: FAILS)
  -> READ-OK: Secret exfiltrated via the API server.
  [RESULT] SUCCEEDED -> NOT as expected: the read went through regardless.
```

  Applying the default-deny policy did not block the read and this shows that `R-P3.1` retrieves the secret's contents purely at the control-plane level, with no dependence on the network side. That does not make **N-F1** irrelevant to the complete `X3` capability because turning the read into an exfiltration still needs open egress, and the same missing baseline that failed to block the read also leaves egress open by default, the same condition `X3`'s joint predicate requires (Table 7.7). Because of this the cell collapses into `X3` rather than being excluded.

- **Refutation: `X2`.** Section 8.3 covers this case in full. In case C, the pod remained isolated by NetworkPolicy while `exec` still succeeded, since the request goes from the API server directly to the kubelet without ever traversing

the pod network the policy inspects. X2 was withdrawn, the only proposed cross-pattern that did not hold.

This also clarifies the three cells in the **R-F4** row that carry both a composing and a control-plane pattern.

R-P4.5, which allows exec, and R-P4.2, which concerns credential issuance, are control-plane-only patterns and do not compose with the network condition, while R-P4.4 does. These cells are therefore marked COLL→X5/CP in Table 7.1. The same family pairing carries a composing instance through R-P4.4 and an ineffective control-plane instance through R-P4.5 or R-P4.2.

The remaining non-composing cells account for the twenty exclusions in Table 7.1, grouped into three exclusion reasons:

- **Collapse into an existing pattern** (ten cells, marked COLL): seven collapse through a substitution on either the RBAC or the network side, while the three cells in the **R-F4** row additionally carry the control-plane instance described above.
- **Borderline substrate** (eight cells, marked SUB): these correspond to borderline substrate conditions, analysed rather than reproduced in the lab, for the same reason X10 is kept outside the core cross-domain patterns.
- **Network-only capability** (two cells, marked NET): the relevant capability is obtained by the network side alone, without requiring an RBAC action, as validated by the reversed test on **R-F1**×**N-F4**.

Overall, the laboratory experiments served two complementary purposes. The eight retained patterns were validated directly, by evaluating each composition under three configurations, both conditions present and then each condition removed in turn, so that removing either side had to fail for the expected reason. The reversed tests, in contrast, examined representative non-composing mechanisms, to ground the analysis of Section 7.4 rather than requiring a dedicated experiment for every cell.

The matrix is defined at the family level, but the final decision to retain or exclude a cell is made by looking at the specific patterns that can instantiate it. Representative tests validate these mechanisms where a test is sufficient, without implying that every excluded cell was independently reproduced.

Ultimately, experimental validation confirmed all eight retained patterns without exception, each ran in three configurations, both conditions present and then each side removed in turn, and the two reduced configurations had to fail for the expected reason for the pattern to remain valid.

Table 8.1: Practical validation of the cross-domain matrix. The last two columns give the printed outcome with each side fixed. Appendix C has the full results for the rows not shown in Section 8.2.

Pattern	Composition	Design	RBAC fixed (B)	Network fixed (C)
X1	**R-F1** × **N-F1**	A/B/C	read refused	read ok, peer unreachable
X3	**R-F3** × **N-F2**	A/B/C	read refused	read ok, send blocked
X4	**R-F1** × **N-F2**	A/B/C	token useless, HTTP 403	token valid, send blocked
X5	**R-F4** × **N-F2**	A/B/C	create pod, HTTP 403	pod created, egress blocked
X6	**R-F4** × **N-F4**	A/B/C	create pod, HTTP 403	pod created, metadata unreachable
X7	**R-F3** × **N-F4**	A/B/C	ConfigMap read refused	address read, endpoint unreachable
X8	**R-F2** ⇒ **N-F3**	precond. + A/B	policy edit refused, HTTP 403	—
X9	**R-F2** ⇒ **N-F6**	precond. + A/B	policy delete refused, HTTP 403	—
Withdrawn pattern and confirmed exclusion				
X2	**R-F4** × **N-F1**†	A/B/C	exec refused	**exec still succeeds** → withdrawn
R-F1 × **N-F4**	(metadata)	reversed	**still reached**	not reached
Reclassified after the run				
R-F3 × **N-F1**	(secret read)‡	reversed	not read	**still read**
X10	**R-F5** × **N-F5**	—	borderline, not run	

† tests R-P4.5 only. The cell itself composes through R-P4.4 instead (Section 8.3).

‡ confirms the read needs no network. It does not exclude the cell, which collapses into X3 instead (Table 7.7).

8.4 Threats to validity

The lab reproduces the mechanisms only, so anything a production environment would add on top of that is not reproduced here. For example, the cloud metadata endpoint (169.254.169.254) has no equivalent in a local cluster, so it is an in-cluster pod at a fixed address and the network fix blocks egress to that address the way blocking the metadata CIDR would. The "external" exfiltration endpoint is an in-cluster collector for the same reason. X10 was not reproduced at all. It is a substrate condition, with the authoriser disabled and the CNI not enforcing. Reproducing it means reconfiguring the control plane rather than composing an attacker capability. None of this changes the joint necessity outcome, which depends only on whether removing one plane breaks the chain.

8.5 Summary

Of the ten cross-patterns originally proposed, eight behave in a running cluster the way the taxonomy says they should, X2 did not and was withdrawn and X10 was left aside because it describes a condition of the substrate rather than an action that can be reproduced. Together with the reversed test that confirmed **R-F1**×**N-F4**'s exclusion and the reclassification of **R-F3**×**N-F1** into X3 once its own missing baseline was checked against X3's predicate, this means that every cell of Table 7.1 now rests either on a lab or on a reason stated in the text, which is what the chapter set out to obtain. The code X2 is left unused in the numbering, so that a reader

comparing the two chapters sees that something was proposed and then dropped, instead of wondering whether a pattern went missing.

Chapter 9

Conclusions

9.1 What was built

This thesis set out to catalogue Kubernetes misconfigurations across two configuration planes and, on top of them, the layer where the two interact. Three taxonomies came out of that work.

Table 9.1 summarises the three. The RBAC side (Chapter 5) and NetworkPolicy (Chapter 6) each follow the same bottom-up method (Chapter 4) on their own corpus.

Taxonomy	Collected	Kept	Families	Patterns
RBAC	381	196	5	17
NetworkPolicy	124	78	6	15
Cross-domain	-	-	5	9

Table 9.1: Collected and kept entries, families and patterns for each taxonomy; cross-domain composes the other two rather than drawing on its own corpus.

The first two taxonomies were built with the same method and cut along the same line, since families in both single-domain taxonomies are defined by the security property that breaks, while underlying mechanisms reside at lower levels. The cross-domain taxonomy depends on this structural alignment, as pairing an RBAC family with a NetworkPolicy family is meaningful only if both models share the same level of abstraction. The cross-domain taxonomy is then organised differently, by the capability a pair unlocks rather than by the property it breaks, for the reason given in Section 7.3 and it stops at two levels instead of three, because each cross-pattern already names the two single domain patterns it is made of and there is nothing left for a third level to hold. What the three share is the way every entry is presented, which is why they are meant to be read as one catalogue rather than as three lists sitting next to each other.

9.2 What the results say

Four things came out of the work:

1. **The dominant failures are the plainest ones**: within NetworkPolicy, **N-F1** (missing isolation) and **N-F5** (substrate not enforcing) together hold 24 of the 40 distinct entries, so what the tools look for most is not a subtly wrong rule but no rule at all, or a rule that selects nothing. RBAC is done the same way, with identity overexposure and broad permissions far more populated than the more specific escalation paths. Building the catalogue from what tools actually check is what makes this visible, whereas starting from intuition would probably have given more room to the sophisticated cases.

2. **The cross-domain layer stays narrow**: of the thirty family-level cells in the composition matrix, nine patterns occupy ten and the other twenty do not compose for one of three reasons. The interaction between the two planes is therefore not everywhere, which is good news for whoever has to defend a cluster, since there are not thirty combinations to worry about. Where it does exist it is concentrated, because the densest column is **N-F2** (over-permissive connectivity), which turns almost any RBAC capability into a way out of the cluster, and exfiltration and lateral movement end up being the two capabilities the layer most often unlocks.

3. **Validation found one pattern that did not hold**: eight cross-patterns behaved in a cluster exactly as the taxonomy predicted, while `X2` did not, because with the target isolated by a NetworkPolicy the `exec` still worked, being served by the API server and never travelling as pod-to-pod traffic. That pattern was withdrawn: `exec`'s own instance contributes nothing to its cell, which instead composes through pod creation into a different cross-pattern, `X5`. The withdrawal matters for the other eight too: it shows the tests could have failed, and did not.

4. **Some regions of the surface are unexamined by tooling**: family **N-F6**, which covers CNI policy CRDs and `AdminNetworkPolicy`, carries zero entries out of 124, and no surveyed scanner checks those objects even though two independent CVEs (`CVE-2023-41333`, `CVE-2026-25804`) and a working lab (`X9`) all show that the exposure is there. The same gap appears again, and wider, in the cross-domain layer, where every tool in the survey looks at one plane at a time, so a weakness that exists only in the combination of two configurations is invisible to all of them.

9.3 Limits

The results of the work carry some limits that follow from how they were produced. What was collected is bound by the tools used for the two single-domain taxonomies, which is why several patterns are stated on analytical grounds rather than derived from a detector, with no empirical backing beyond the documentation and, in some cases, a CVE. The practical validation runs on `kind` with Calico, so it shows that the mechanisms work rather than how often they occur in production, since a different CNI could enforce differently, and two of its elements, the cloud metadata endpoint and the external collector, are simulated as in-cluster pods at fixed addresses rather than the real services they stand in for. `X10` was analysed in theory rather than run for the same kind of reason, because it describes a condition of the substrate rather

than a capability an attacker exercises. The joint necessity criterion is also applied at family level, so two families may compose while a specific pair of subpatterns beneath them does not, which means the matrix says where an interaction is possible rather than that every instance of it is exploitable.

9.4 Future work

This work can be extended in several directions:

1. The most immediate one is a cross-domain detector, a tool that should evaluate the two planes together in real repositories to help locate vulnerabilities of this kind. Building it would mean reading RBAC and NetworkPolicy objects from the same cluster or the same set of manifests, deriving a capability for each ServiceAccount and then evaluating the predicates over the pair. The main obstacle is that manifests in real repositories rarely carry all the information a detector needs, because they are often defined outside the repository that holds the workload, so a step that collects the data and normalises it has to come first. Beyond detection, automated remediation of single domain policy misconfigurations is already a well-established line of work. Firewall anomalies have been atomised and resolved automatically [6, 10], verifiable and optimised policies have been enforced in SDN-aware IoT networks [13] and detected attacks have been countered through autonomous firewall reconfiguration [9, 8], down to sustainable configuration across the edge-to-cloud continuum [12]. A broader survey covers network security configuration automation as a whole [7]. The same automation line has also been extended to security orchestration across multiple clusters [11], the natural setting once a cross-domain detector has to reason about ServiceAccounts and NetworkPolicies spanning more than one cluster. Extending this kind of remediation to the cross-domain case is a natural continuation once a detector can locate the affected pair.

2. Secondly, another possible direction would be to measure how often this kind of misconfiguration occurs in real repositories and to analyse the resulting data.

3. Thirdly, a complementary direction is prevention rather than detection. Intent-based specification lets an operator describe the desired isolation directly and derive the underlying `NetworkPolicy` objects from that intent instead of authoring them by hand, an approach proposed both within a single cluster [46] and across the edge-to-cloud continuum [47]. It would remove the manual specification step where several of the patterns collected here, **N-F2** most of all, originate.

4. Lastly, another possible direction would be to extend the current methodology to also include Pod Security and admission control, which were deliberately kept out of scope, though some patterns already sit on the boundary of Pod Security.

Moreover, the labs cover eight cross-patterns, the withdrawn `X2`, the confirmed exclusion of **R-F1**×**N-F4** and the reclassification of **R-F3**×**N-F1** into `X3`, and the

cells that are still argued in prose could be anchored the same way, with one lab for each of the three reasons rather than one for each cell. `X10` waits on a different setup, with the authoriser and the CNI reconfigured, before it can leave the borderline set in either direction and **N-F6** waits on the tools, so that its patterns may one day be derived from a corpus like all the others. Both gaps come from the same cause that motivated this work in the first place, since nobody has looked there yet.

Appendix A

Family N-F6 — CNI-extension & Admin Policy

No entries are assigned to this family, because the CNI-extension and `AdminNetworkPolicy` space is barely covered by existing tooling. Mapping this under-tooled region is what this appendix adds. The family is presented apart from Chapter 6 for that reason, rests on analysis rather than on corpus evidence, and concerns objects (CNI CRDs, ANP/BANP) outside the standard `NetworkPolicy` API. It remains part of the taxonomy, counted in Table 6.1, and cross-pattern `X9` composes with it. The absence of tool entries is not the same as an absence of evidence: `N-P6.1` and `N-P6.2` are each anchored to a published, independently verifiable CVE (Appendix A.1, Appendix A.2), which is what justifies keeping the family in the taxonomy rather than treating it as a hypothetical gap. `N-P6.3` remains stated on analytical grounds alone, without a CVE of its own.

Effect. A single cluster-wide or admin policy reopens connectivity that every namespaced policy denied (ImpactScope: cluster).

Family definition

Family **N-F6** brings together misconfigurations specific to the policy mechanisms that go beyond the standard namespaced `NetworkPolicy`, the cluster-scoped CRDs of CNI plug-ins (Calico `GlobalNetworkPolicy` and tiers, Cilium `CiliumClusterwideNetworkPolicy`) and the upstream `AdminNetworkPolicy` (ANP) / `BaselineAdminNetworkPolicy` (BANP) API. These objects introduce two capabilities absent from standard NetworkPolicy, cluster-wide scope and explicit precedence, and with them two new failure classes, an over-broad cluster-wide override and a precedence/ordering mistake that lets an `Allow` shadow a `Deny`.

Because no surveyed scanner focuses on these checks, all three patterns are stated on analytical grounds rather than discovered from a tool rule, though `N-P6.1` and `N-P6.2` are each anchored to a published CVE below.

Threat narrative

Admin and cluster-wide policies are evaluated before ordinary NetworkPolicy and can override it. An over-broad `Allow` at this tier (`N-P6.1`) silently reopens connectivity that every namespaced policy tried to deny (a single object undoing the whole isolation posture). A precedence mistake (`N-P6.2`) produces the same effect through ordering rather than width. An `Allow` placed in a tier evaluated before a `Deny`, or a misordered ANP/NP/BANP stack, wins over the intended denial. An over-permissive baseline (`N-P6.3`) sets the cluster's default too loose, weakening every namespace that does not override it.

MITRE ATT&CK mapping. In MITRE ATT&CK for Containers, a cluster-wide or admin policy that silently overrides namespaced isolation sits under the Defense Impairment tactic and, once isolation is gone, opens the way to Lateral Movement.

Technique	Why it applies
Disable or Modify Tools T1685	a single cluster-scoped or admin object overrides every namespaced policy beneath it

General detection criterion

Detection criterion — family N-F6

A cluster-scoped or admin policy (Calico `GlobalNetworkPolicy`/tier, `CiliumClusterwideNetworkPolicy`, ANP/BANP) admits connectivity that the namespaced policy layer denies, through either (i) over-broad scope (`N-P6.1`), (ii) precedence/ordering (`N-P6.2`), or (iii) a too-permissive baseline (`N-P6.3`).

Patterns of family N-F6

Pattern	Name	L3	Entries
`N-P6.1`	Clusterwide override too broad	—	0 †
`N-P6.2`	Precedence Inversion	2	0 †
`N-P6.3`	Admin baseline too permissive	—	0 †

† stated on analytical grounds; no supporting detector.

A.1 N-P6.1 — Clusterwide override too broad

Pattern definition. A cluster-scoped policy (`CiliumClusterwideNetworkPolicy`, Calico `GlobalNetworkPolicy`) with an over-broad `Allow` that applies across all namespaces, overriding the namespaced default-deny baselines beneath it.

Detection criterion — N-P6.1

A cluster-scoped CNI policy contains an `Allow` whose endpoint/peer selector is empty or cluster-wide, re-admitting traffic denied at the namespace layer.

Collected entries. None. No surveyed tool checks cluster-wide CNI overrides, so the pattern is stated on analytical grounds.

✗ **Vulnerable manifest** (triggers N-P6.1)

```
# A cluster wide Allow with an empty selector re-opens traffic for EVERY pod,
# overriding the namespaced default-deny baselines.
apiVersion: cilium.io/v2
kind: CiliumClusterwideNetworkPolicy
metadata:
  name: allow-all-cluster
spec:
  endpointSelector: {}          # <-- every endpoint, cluster wide (N-P6.1)
  ingress:
    - {}                        # allow from anywhere
```

Real-world evidence (CVE)

Manually verified: `CVE-2023-41333` [35] (CVSS 6.9). An attacker able to create or modify `CiliumNetworkPolicy` objects in one namespace can affect traffic across the entire Cilium cluster, bypassing policy enforcement in other namespaces (exactly the cluster-wide override risk of `N-P6.1` and the realisation of cross-pattern `X9`).

Remediation. Restrict cluster-wide policies to narrowly scoped endpoint selectors and explicit peers and reserve them for genuinely cluster wide rules such as allowing DNS rather than for broad `Allow`s that reopen what the namespaced layer already denies. `CVE-2023-41333` is a reminder of how far a mistake here reaches, since a change made from inside one namespace was able to affect Cilium enforcement across the whole cluster. Because a cluster-wide policy sits above every namespaced default-deny baseline, even a narrow mistake at this tier has cluster-wide consequences, which is why access to create or edit these objects should stay limited to cluster administrators.

Enforcement policy

```
# Corrected: a cluster wide policy scoped to one purpose (DNS), not allow-all.
apiVersion: cilium.io/v2
kind: CiliumClusterwideNetworkPolicy
metadata:
  name: allow-dns-only
spec:
  endpointSelector: {}
  egress:
    - toEndpoints:
        - matchLabels:
            k8s:io.kubernetes.pod.namespace: kube-system
      toPorts:
        - ports: [{ port: "53", protocol: UDP }]
```

A.2 N-P6.2 — Precedence Inversion

Pattern definition. A precedence or ordering mistake among layered policies. The two policy stacks that support layering order their rules differently, so the mistake takes two forms.

Detection criterion — N-P6.2

An `Allow` rule sits at a tier/priority evaluated before a `Deny` that targets an overlapping peer set, so the `Allow` takes precedence and the `Deny` never applies.

Real-world evidence (CVE)

Manually verified: `CVE-2026-25804` [41] (CVSS 8.0). Antrea's network-policy priority-assignment system has a `uint16` arithmetic overflow that produces incorrect OpenFlow priorities when many policies are handled. This is a real precedence bug that makes the wrong rule win, which is exactly the failure mode `N-P6.2` describes.

Collected entries. None from the surveyed tools, so the pattern is stated on analytical grounds. Only a real CVE anchors it (above).

L3 — N-P6.2.1 Upstream ANP–NP–BANP order

`AdminNetworkPolicy` (ANP) is evaluated before namespaced `NetworkPolicy`, which is evaluated before `BaselineAdminNetworkPolicy` (BANP). An `Allow` placed at the ANP tier therefore wins over a namespaced `Deny`, contrary to the author's intent. The namespace owner writes a denial and the cluster administrator, often without meaning to, has already allowed the traffic.

✗ **Vulnerable manifest** (triggers N-P6.2.1)

```yaml
# An ANP Allow at high priority overrides the namespaced NetworkPolicy Deny:
# ANP is evaluated BEFORE namespaced NP, so this Allow wins (N-P6.2.1).
apiVersion: policy.networking.k8s.io/v1alpha1
kind: AdminNetworkPolicy
metadata:
  name: tenant-allow
spec:
  priority: 10
  subject:
    namespaces: {}
  ingress:
    - action: Allow            # <-- evaluated before namespaced Deny (N-P6.2.1)
      from:
        - namespaces: {}
```

L3 — N-P6.2.2 Calico tier misordering

In Calico, policies are grouped into ordered tiers, and an `Allow` placed in an earlier tier wins over a more specific `Deny` placed later, ending the evaluation before that later tier is ever reached. The `Pass` action exists precisely to hand the decision to the next tier, and a rule that allows where it should pass ends the evaluation on the spot.

✗ **Vulnerable manifest** (triggers N-P6.2.2)

```yaml
# The 'platform' tier is evaluated before 'app'. This Allow ends evaluation,
# so the stricter Deny written in the later tier is never reached (N-P6.2.2).
apiVersion: crd.projectcalico.org/v1
kind: GlobalNetworkPolicy
metadata:
  name: platform.allow-internal
spec:
  tier: platform
```

```
  order: 100
  selector: all()
  ingress:
    - action: Allow             # <-- should have been Pass (N-P6.2.2)
```

Remediation. Model the full evaluation order (ANP → NP → BANP, or Calico tier order) when authoring rules. Keep broad `Allow`s out of high-priority tiers and reserve those tiers for `Deny`/guardrail rules. For the Calico tier case, the fix is a single action. The early tier should pass the decision on instead of ending the evaluation there.

Enforcement policy

```
# Corrected: the 'platform' tier passes the decision to the next tier instead
# of ending evaluation, so the Deny written in the later 'app' tier is
# still reached.
apiVersion: crd.projectcalico.org/v1
kind: GlobalNetworkPolicy
metadata:
  name: platform.allow-internal
spec:
  tier: platform
  order: 100
  selector: all()
  ingress:
    - action: Pass              # hands the decision to the next tier
```

A.3 N-P6.3 — Admin baseline over-permissive

Pattern definition. The `BaselineAdminNetworkPolicy` (the cluster's default when no other policy matches) is set too permissively (for example, a baseline `Allow`), so every namespace that does not explicitly override it inherits open connectivity. A loose baseline silently weakens the whole cluster.

Detection criterion — N-P6.3

The `BaselineAdminNetworkPolicy` default action is `Allow` (or broadly permissive) rather than a deny-oriented guardrail.

Collected entries. None. No surveyed tool checks the `BaselineAdminNetworkPolicy` default action, so the pattern is stated on analytical grounds.

✗ Vulnerable manifest (triggers N-P6.3)

```
# A permissive BANP makes the cluster wide default allow-all, so any
# namespace without its own policy inherits open ingress (N-P6.3).
apiVersion: policy.networking.k8s.io/v1alpha1
kind: BaselineAdminNetworkPolicy
metadata:
  name: default
spec:
  subject:
    namespaces: {}
  ingress:
    - action: Allow             # <-- permissive cluster default (N-P6.3)
      from:
        - namespaces: {}
```

Remediation. Set the BANP's ingress fallback to `Deny` instead of `Allow`, so a namespace with no policy of its own starts closed rather than open, mirroring the

default-deny principle of **N-F1** at cluster scope. Because the BANP applies only where no other policy has already decided the traffic, tightening it strengthens every namespace that never wrote its own rule, without touching the namespaces that already override it explicitly.

Enforcement policy

```yaml
# Corrected: the cluster default is now Deny, so namespaces start closed.
apiVersion: policy.networking.k8s.io/v1alpha1
kind: BaselineAdminNetworkPolicy
metadata:
  name: default
spec:
  subject:
    namespaces: {}
  ingress:
    - action: Deny            # <-- safe cluster wide fallback (N-P6.3)
      from:
        - namespaces: {}
```

Appendix B

Manually verified CVEs

A CVE is attached to a pattern only after being read by hand against the official description, because automatic keyword matching pins a handful of generic high severity Kubernetes CVEs onto almost everything (Chapter 4). This appendix collects that work in one place: every CVE checked, the pattern it ended up on and what it actually shows.

It also records what the pattern boxes of Chapter 5 and Chapter 6 leave out. Several CVEs passed the check but are not quoted anywhere, because the pattern they support is already established by other evidence. They are listed here rather than dropped.

B.1 RBAC domain

The four Windows node escalations (`CVE-2023-3676`, `-3893`, `-3955`, `-5528`) show why the list is kept here rather than in the chapter. They are four CVEs for one capability, namely creating a pod on a node the attacker should not control, which `R-P4.4` already covers. Putting all four in the pattern box would make it look like four separate weaknesses. Dropping them would hide that `R-P4.4` is among the best-attested patterns overall.

Table B.1: CVEs verified for the RBAC taxonomy. "In text" marks the ones quoted in a pattern box, while the others are recorded here only.

CVE	CVSS	Pattern	What it shows	In text
CVE-2018-1002102 [20]	2.6	R-P4.6	an already-compromised kubelet returns a redirect the API server follows with its own client-cert credentials: evidence for the substrate-proxy trust surface R-P4.6.1 sits on, not an instance of the RBAC criterion, since the attacker here starts from node compromise rather than a granted nodes/proxy rule	
CVE-2018-1002105 [21]	9.8	R-P4.5	a proxied upgrade request keeps the connection open, and further requests over it reach the kubelet authenticated with the API server's own credentials rather than the caller's, so exec/attach/ portforward became reachable without holding R-P4.5 at all; recorded here as evidence of how severe that capability class is, not as an instance of the RBAC criterion	
CVE-2019-11243 [22]	8.1	R-P1.3	AnonymousClientConfig() returned a config that still carried the bearer token	✓
CVE-2019-11247 [23]	8.1	R-P2.1	a cluster-scoped custom resource is reachable through a namespaced request and is authorised by the namespace's roles	✓
CVE-2020-8562 [24]	2.2	R-P4.6	a TOCTOU in address validation lets an already-proxied connection still reach link-local and localhost addresses, defeating the mitigation meant to block them: it bypasses a safeguard around R-P4.6.1/R-P4.6.2 rather than the RBAC grant itself	
CVE-2021-25740 [25]	3.1	R-P2.3	a confused deputy sends traffic to destinations the sender could not otherwise reach: the built-in aggregated roles allow it	✓
CVE-2022-3294 [26]	6.6	R-P4.6	a subject able to modify a Node's registered address, combined with R-P4.6.1's nodes/proxy permission, redirects the API server's proxy traffic to secure endpoints on the control plane network	
CVE-2023-2728 [28]	6.5	R-P3.1	the mountable-secrets policy of the ServiceAccount admission plugin is bypassed	✓
CVE-2023-2878 [30]	6.5	R-P1.3	secrets-store-csi-driver writes ServiceAccount tokens to its logs	✓
CVE-2023-3676 [31]	8.8	R-P4.4	creating a pod on a Windows node escalates to administrator on that node	
CVE-2023-3893 [32]	8.8	R-P4.4	the same escalation through kubernetes-csi-proxy	
CVE-2023-3955 [34]	8.8	R-P4.4	a further variant of the same Windows-node escalation	
CVE-2023-5528 [36]	7.2	R-P4.4	creating pods and persistent volumes on a Windows node escalates to administrator	
CVE-2024-3177 [37]	2.7	R-P3.1	a second bypass of the mountable-secrets policy, through envFrom	✓
CVE-2024-31989 [38]	9.1	—	in Argo CD, an unprivileged pod in another namespace reaches Redis because the CNI does not enforce policy: an RBAC-side impact produced by a network condition	✓
CVE-2025-7445 [40]	6.5	R-P1.3	secrets-store-sync-controller writes ServiceAccount tokens to its logs	✓

B.2 NetworkPolicy domain

Table B.2: CVEs verified for the NetworkPolicy taxonomy. "In text" marks the ones quoted in a pattern box, while the others are recorded here only.

CVE	CVSS	Pattern	What it shows	In text
CVE-2023-27595 [29]	6.5	N-P5.1	in Cilium 1.13.0 the eBPF programs are not yet attached at start-up, so for a short window the node enforces no policy at all	✓
CVE-2023-39347 [33]	7.6	N-P2.1	Cilium trusts the labels supplied on a pod update, so an attacker who can set labels has the wrong policies applied to the pod	✓
CVE-2023-41333 [35]	6.9	N-P6.1	creating or modifying a `CiliumNetworkPolicy` in one namespace affects traffic across the whole cluster	✓
CVE-2024-7598 [39]	3.1	N-P5.1	during namespace termination the policies can be deleted before the pods they protect, leaving a window with pods running unprotected	✓
CVE-2026-25804 [41]	8.0	N-P6.2	an arithmetic overflow in Antrea's priority assignment produces incorrect OpenFlow priorities, so the wrong rule wins	✓
CVE-2026-32768 [42]	7.9	**N-F2**	a miswritten `NetworkPolicy` lets an actor pivot to pods outside its origin namespace: an over-broad rule, not a missing one	✓
CVE-2026-33726 [43]	5.4	N-P5.1	in Cilium, ingress policy is not enforced for pod-to-L7-Service traffic with a node-local backend under per-endpoint routing	✓

The two tables define those numbers. There are seventeen verified CVEs on the RBAC side and seven on the NetworkPolicy side, five of which are defects in a CNI implementation rather than in the `NetworkPolicy` API itself. This is the point already made in Chapter 6. The serious NetworkPolicy failures are configuration failures and the CVE system catalogues software defects, so it picks up the substrate and misses the configuration.

Appendix C

The validation lab environment

Chapter 8 calls the practical validation a reproducible lab without saying what building it required. This appendix gives the missing part: the cluster, the folder layout and the shared code that every one of the eleven labs runs on, so that the setup can actually be reproduced and not only read about.

Cluster

All labs share a single `kind` cluster, created once and reused by every pattern in its own namespace. The default CNI, `kindnet`, does not enforce NetworkPolicy at all, which would make every network side fix pass without doing anything, so the cluster disables it and installs Calico instead. `kind-config.yaml` sets `disableDefaultCNI: true` and a pod subnet matching Calico's default range and `setup-cluster.sh` creates the cluster from that config, installs Calico and waits for it and CoreDNS to be ready before handing back control:

Cluster setup

```
bash setup-cluster.sh       # once: kind cluster + Calico, shared by all labs
...
bash resetCluster.sh        # at the end: tears the cluster down
```

Layout

`CrossValidator/` holds one folder per retained pattern (`X1-...`, `X3-...` and so on), plus an `Excluded-...` folder for each of the two reversed-outcome checks and a third, `Excluded-RF4xNF1-exec`, for the withdrawn `X2` composition. Every folder follows the same shape:

Layout of one lab folder

```
X3-secret-egress/
|-- README.md              what the lab shows and how to read the result
|-- run-validation.sh      orchestrates cases A/B/C (or precond+A/B)
|-- attack.sh              the exploit, run from inside the compromised pod
`-- manifests/
    |-- base/               namespace, Secret, victim pod, collector
    |-- rbac/                the weak Role/ClusterRole and its fixed version
    `-- network/             the NetworkPolicy fix
```

Three folder shapes exist, matching the three test designs of Chapter 8: **compositions** run the full A/B/C comparison, **disabling patterns** run a precondition

check plus cases A/B and **excluded cells** run the same A/B/C comparison with the roles of success and failure reversed.

Shared helpers

Every `run-validation.sh` sources `lib/validation-lib.sh`, so the orchestration logic is written once instead of copied and pasted into ten scripts. The functions that matter are:

- `apply` — a quiet `kubectl apply`.
- `reset_lab` — clears leftover NetworkPolicies before a run starts. Policies are additive, so one left over from a previous run on the shared cluster would silently add itself to the current case.
- `pod_ip` — reads a pod's IP and fails loudly if it is not `Running`, instead of writing an empty address into a policy and producing a result that means nothing.
- `RUN_ID` — a per-run identifier that every exfiltration marker carries, so a log line from an earlier run can never be mistaken for one from the current case.
- `sa_api_code` / `sa_api_patch` — call the API server from inside the pod with the ServiceAccount token mounted in it, which is the only credential a real attacker in that pod would have.
- `wait_denied` — waits for an RBAC revocation to actually take effect before running a case that depends on it, since the authoriser does not apply a revoked binding instantly.
- `run_caso` / `print_results` (and the `_bridge` / `_excluded` variants) — run one case, collect its outcome and print the results table at the end in the form matched against in Chapter 8.

Running a lab

Typical run

```
bash setup-cluster.sh                    # once
cd X3-secret-egress
bash run-validation.sh
```

Each case prints what it did (an API read, an exfiltration attempt, an RBAC check) and whether it succeeded or failed. At the end, `run-validation.sh` prints a table with one row per case, the expected outcome, the observed one and whether they match, followed by a verdict line that states whether joint necessity holds. That table, translated, is what Section 8.2 and Table 8.1 report for every pattern.

Results for every pattern

`X1` and `X8` are reported in full in Section 8.2, and the `X2` refutation in Section 8.3, with the exact output each case printed. The remaining patterns are only summarised in Table 8.1. This section closes that gap by reproducing what their own runs printed, translated and compressed into the same table each script prints on its own.

Lab results — compositions

```
X3 secret-egress
  A complete        SUCCEEDS / SUCCEEDS   B RBAC fixed   FAILS / FAILS
  C network fixed   FAILS / FAILS
  => joint necessity demonstrated, a valid cross-pattern.

X4 token-exfiltration
  A complete        SUCCEEDS / SUCCEEDS   B RBAC fixed   FAILS / FAILS
  C network fixed   FAILS / FAILS
  => joint necessity demonstrated, a valid cross-pattern.

X5 pod-egress-channel
  A complete        SUCCEEDS / SUCCEEDS   B RBAC fixed   FAILS / FAILS
  C network fixed   FAILS / FAILS
  => joint necessity demonstrated, a valid cross-pattern.

X6 pod-metadata
  A complete        SUCCEEDS / SUCCEEDS   B RBAC fixed   FAILS / FAILS
  C network fixed   FAILS / FAILS
  => joint necessity demonstrated, a valid cross-pattern.

X7 endpoint-disclosure
  A complete        SUCCEEDS / SUCCEEDS   B RBAC fixed   FAILS / FAILS
  C network fixed   FAILS / FAILS
  => joint necessity demonstrated, a valid cross-pattern.
```

Lab results — disabling pattern and excluded cells

```
X9 wildcard-cni-policy   (precondition confirmed: target blocked)
  A disable         SUCCEEDS / SUCCEEDS   B RBAC fixed   FAILS / FAILS
  => enabling bridge demonstrated. The RBAC permission has effect only
     because it acts on an active network control.

Excluded R-F1 x N-F4 (metadata)
  A complete        SUCCEEDS / SUCCEEDS   B RBAC fixed   SUCCEEDS / SUCCEEDS
  C network fixed   FAILS / FAILS
  => exclusion demonstrated. The RBAC side is redundant, the capability
     is single-domain.

R-F3 x N-F1 (secret) -- reclassified, not an exclusion
  A complete        SUCCEEDS / SUCCEEDS   B RBAC fixed   FAILS / FAILS
  C network fixed   SUCCEEDS / SUCCEEDS
  => the read needs no network, confirming R-P3.1 is a control-plane
     call. It does not show the cell is single-domain: the same missing
     baseline also defaults egress to allow-all, which is what the cell
     now composes into (X3, Table 7.7). Case C only re-ran the read,
     not the send this collapse depends on.
```

Every case above matched its prediction, as do those of `X1` and `X8` reported in Section 8.2. The sole exception is `X2` (Section 8.3), whose mismatch is exactly what led to its withdrawal from the taxonomy.

Bibliography

[1] Aqua Security. kube-bench — cis kubernetes benchmark. `https://github.com/aquasecurity/kube-bench`, 2025.

[2] Aqua Security. trivy-checks. `https://github.com/aquasecurity/trivy-checks`, 2025.

[3] ARMO. Kubescape. `https://kubescape.io/`, 2025.

[4] Ahmet Alp Balkan. Kubernetes networkpolicy recipes. `https://github.com/ahmetb/kubernetes-network-policy-recipes`, 2025.

[5] Bridgecrew. Checkov. `https://www.checkov.io/`, 2025.

[6] Daniele Bringhenti, Simone Bussa, Riccardo Sisto, and Fulvio Valenza. Atomizing firewall policies for anomaly analysis and resolution. *IEEE Transactions on Dependable and Secure Computing*, 22(3):2308–2325, 2025.

[7] Daniele Bringhenti, Guido Marchetto, Riccardo Sisto, and Fulvio Valenza. Automation for network security configuration: State of the art and research trends. *ACM Computing Surveys*, 56(3):57:1–57:37, 2024.

[8] Daniele Bringhenti, Francesco Pizzato, Riccardo Sisto, and Fulvio Valenza. A looping process for cyberattack mitigation. In *2024 IEEE International Conference on Cyber Security and Resilience (CSR)*, pages 276–281, 2024.

[9] Daniele Bringhenti, Francesco Pizzato, Riccardo Sisto, and Fulvio Valenza. Autonomous attack mitigation through firewall reconfiguration. *International Journal of Network Management*, 35(1):1–18, 2025.

[10] Daniele Bringhenti, Lucia Seno, and Fulvio Valenza. An optimized approach for assisted firewall anomaly resolution. *IEEE Access*, 11:119693–119710, 2023.

[11] Daniele Bringhenti, Riccardo Sisto, and Fulvio Valenza. Security automation for multi-cluster orchestration in kubernetes. In *2023 IEEE 9th International Conference on Network Softwarization (NetSoft)*, 2023.

[12] Daniele Bringhenti and Fulvio Valenza. Greenshield-e2c: A sustainable energy-aware firewall configuration mechanism for edge-to-cloud continuum. *Computer Networks*, 2026.

[13] Daniele Bringhenti, Jalolliddin Yusupov, Alejandro Molina Zarca, Fulvio Valenza, Riccardo Sisto, Jorge Bernal Bernabe, and Antonio Skarmeta. Automatic, verifiable and optimized policy-based security enforcement for sdn-aware iot networks. *Computer Networks*, 213:109123, 2022.

[14] Checkmarx. Kics — keeping infrastructure as code secure. https://kics.io/, 2025.

[15] CVE Program. Cve list v5. https://github.com/CVEProject/cvelistV5, 2025.

[16] Kubernetes SIG-Network. Kubernetes network policy api (adminnetworkpolicy / baselineadminnetworkpolicy). https://github.com/kubernetes-sigs/network-policy-api, 2025.

[17] Kyverno. Kyverno — kubernetes native policy management. https://kyverno.io/, 2025.

[18] MITRE. Mitre att&ck for enterprise, version 19.1 (containers platform). https://attack.mitre.org/versions/v19/matrices/enterprise/containers/, 2026.

[19] NIST. National vulnerability database (nvd). https://nvd.nist.gov/, 2025.

[20] NVD. Cve-2018-1002102. https://nvd.nist.gov/vuln/detail/CVE-2018-1002102, 2018.

[21] NVD. Cve-2018-1002105. https://nvd.nist.gov/vuln/detail/CVE-2018-1002105, 2018.

[22] NVD. Cve-2019-11243. https://nvd.nist.gov/vuln/detail/CVE-2019-11243, 2019.

[23] NVD. Cve-2019-11247. https://nvd.nist.gov/vuln/detail/CVE-2019-11247, 2019.

[24] NVD. Cve-2020-8562. https://nvd.nist.gov/vuln/detail/CVE-2020-8562, 2020.

[25] NVD. Cve-2021-25740. https://nvd.nist.gov/vuln/detail/CVE-2021-25740, 2021.

[26] NVD. Cve-2022-3294. https://nvd.nist.gov/vuln/detail/CVE-2022-3294, 2022.

[27] NVD. Cve-2022-46167. https://nvd.nist.gov/vuln/detail/CVE-2022-46167, 2022.

[28] NVD. Cve-2023-2728. https://nvd.nist.gov/vuln/detail/CVE-2023-2728, 2023.

[29] NVD. Cve-2023-27595. https://nvd.nist.gov/vuln/detail/CVE-2023-27595, 2023.

[30] NVD. Cve-2023-2878. https://nvd.nist.gov/vuln/detail/CVE-2023-2878, 2023.

[31] NVD. Cve-2023-3676. https://nvd.nist.gov/vuln/detail/CVE-2023-3676, 2023.

[32] NVD. Cve-2023-3893. `https://nvd.nist.gov/vuln/detail/CVE-2023-3893`, 2023.

[33] NVD. Cve-2023-39347. `https://nvd.nist.gov/vuln/detail/CVE-2023-39347`, 2023.

[34] NVD. Cve-2023-3955. `https://nvd.nist.gov/vuln/detail/CVE-2023-3955`, 2023.

[35] NVD. Cve-2023-41333. `https://nvd.nist.gov/vuln/detail/CVE-2023-41333`, 2023.

[36] NVD. Cve-2023-5528. `https://nvd.nist.gov/vuln/detail/CVE-2023-5528`, 2023.

[37] NVD. Cve-2024-3177. `https://nvd.nist.gov/vuln/detail/CVE-2024-3177`, 2024.

[38] NVD. Cve-2024-31989. `https://nvd.nist.gov/vuln/detail/CVE-2024-31989`, 2024.

[39] NVD. Cve-2024-7598. `https://nvd.nist.gov/vuln/detail/CVE-2024-7598`, 2024.

[40] NVD. Cve-2025-7445. `https://nvd.nist.gov/vuln/detail/CVE-2025-7445`, 2025.

[41] NVD. Cve-2026-25804. `https://nvd.nist.gov/vuln/detail/CVE-2026-25804`, 2026.

[42] NVD. Cve-2026-32768. `https://nvd.nist.gov/vuln/detail/CVE-2026-32768`, 2026.

[43] NVD. Cve-2026-33726. `https://nvd.nist.gov/vuln/detail/CVE-2026-33726`, 2026.

[44] Open Policy Agent. Opa gatekeeper — policy library. `https://open-policy-agent.github.io/gatekeeper-library/website/`, 2025.

[45] OWASP. Kubernetes security cheat sheet. `https://cheatsheetseries.owasp.org/cheatsheets/Kubernetes_Security_Cheat_Sheet.html`, 2025.

[46] Francesco Pizzato, Daniele Bringhenti, Riccardo Sisto, and Fulvio Valenza. An intent-based solution for network isolation in kubernetes. In *2024 IEEE 10th International Conference on Network Softwarization (NetSoft)*, pages 381–386, 2024.

[47] Francesco Pizzato, Daniele Bringhenti, Riccardo Sisto, and Fulvio Valenza. Intent-driven network isolation for the cloud computing continuum. *Journal of Network and Systems Management*, 34(1):1–39, 2026.

[48] Francesco Pizzato, Daniele Bringhenti, and Fulvio Valenza. A comparative study of authorisation mechanisms in kubernetes-based service platforms. *Applied Cybersecurity & Internet Governance*, 2026.

[49] StackRox. kube-linter. `https://github.com/stackrox/kube-linter`, 2025.

[50] The Falco Authors. Falco. `https://falco.org/`, 2025.

[51] The Kubernetes Authors. Authorization overview. `https://kubernetes.io/docs/reference/access-authn-authz/authorization/`, 2025.

[52] The Kubernetes Authors. Kubelet authentication/authorization. `https://kubernetes.io/docs/reference/access-authn-authz/kubelet-authn-authz/`, 2025.

[53] The Kubernetes Authors. Network policies. `https://kubernetes.io/docs/concepts/services-networking/network-policies/`, 2025.

[54] The Kubernetes Authors. Node authorization. `https://kubernetes.io/docs/reference/access-authn-authz/node/`, 2025.

[55] The Kubernetes Authors. Role based access control good practices. `https://kubernetes.io/docs/concepts/security/rbac-good-practices/`, 2025.

[56] The Kubernetes Authors. Using rbac authorization. `https://kubernetes.io/docs/reference/access-authn-authz/rbac/`, 2025.

[57] The Kubernetes Authors. Using rbac authorization — controller roles. `https://kubernetes.io/docs/reference/access-authn-authz/rbac/#controller-roles`, 2025.

[58] The Kubernetes Authors. Volumes. `https://kubernetes.io/docs/concepts/storage/volumes/`, 2025.