elastisys

Free Guide

How to Security-Harden Your Kubernetes Platform against “unknown unknowns”

elastisys

Table of Contents

Restrictive NetworkPolicies **2**

What are NetworkPolicies? 2

Understanding NetworkPolicies by Example 3

Why are restrictive NetworkPolicies important? 4

How can you protect yourself against the next zero-day vulnerability? 4

How to write the restrictive NetworkPolicies? 4

Final Thoughts 5

Further Reading 5

Vulnerability Management **6**

What are vulnerabilities? 6

Vulnerability Timeline 7

How do you find out about a vulnerability? 8

How to act when you learn about a vulnerability? 9

Final Thoughts 10

Further Reading 10

Intrusion Detection **11**

How does an attack work? 11

What is intrusion detection? How does it work? 13

Rules: The Key to Relevant Intrusion Detection 14

Final Thoughts 16

Further Reading 16

Takeaways **18**

2023! What a rough start. When I opened the email on my first working day (9 Jan), Have I Been Pwned had already notified me of 5 data breaches. Indeed, data breaches are not only getting more frequent, but also more expensive. Needless to say, this puts pressure on those of us who are responsible for ensuring Kubernetes platform security.

In our previous guide (How to Operate a Secure Kubernetes Platform), we discussed basic hygiene for operating a secure and stable Kubernetes Platform. Having scope, alerting culture, disaster recovery training and maintenance, will help you protect your Kubernetes platform against "known knowns". It acts like a solid security foundation.

In this guide we discuss how to security-harden your Kubernetes platform against "known unknowns" and "unknown unknowns" with restrictive NetworkPolicies, vulnerability management and intrusion detection."

Restrictive NetworkPolicies

The phrase "don't talk to strangers" is counterproductive to making new friends, but it's good advice when it comes to security. Several information security standards (e.g., ISO 27001 A13.1) and regulations put great emphasis on network security.

In this section we present NetworkPolicies, how they help improve your security posture and how to write restrictive NetworkPolicies.

What are NetworkPolicies?

NetworkPolicies are Kubernetes's equivalent of firewalls. Firewalls are usually configured to allow or deny traffic from and to specified IP addresses. These IP addresses are most commonly specified as a network prefix (e.g., 192.168.100.0/24 for IPv4 or 2001:DB8::/32 for IPv6). These are also called CIDR, a term which only makes sense if you were a network administrator before 1993 or are really curious about Internet history.

Specifying IP addresses makes sense when workloads are "pet-like" and don't change IP addresses too often. However, Kubernetes wants you to treat containers as "cattles". Indeed, Kubernetes frequently terminates and starts containers, as required to roll out new versions, perform self-healing or scale horizontally. Every time a container is started it receives a new IP address. Needless to say, we need another way to specify the source and destination of allowed traffic.

In fact, the smallest deployable unit in Kubernetes is a Pod. A Pod consists of several containers, which share the same Linux network namespace. This means that containers within the same Pod can talk via localhost and share the same unique IP address when talking to containers in other Pods.

NetworkPolicies are namespaced Kubernetes resources which allow you to control traffic going into (ingress) and going out of (egress) your Pods. Traffic between containers within the same Pod cannot be restricted with NetworkPolicies. If you are familiar with firewalls, be aware that there are three main conceptual differences between them:

1. NetworkPolicies restrict traffic “around” a Pod, not “in” the network. Think of them more like SecurityGroups around virtual machines.
2. NetworkPolicies select Pods by matching their labels. External systems can still be matched by network prefix.
3. NetworkPolicies are stateful. Whether for TCP or UDP, you describe allowed client-to-server traffic. The matching server-to-client traffic is automatically allowed. Firewalls can be both stateful and stateless.

Understanding NetworkPolicies by Example

The code below illustrates an example of a NetworkPolicy. The policy applies to all Pods in the cklein-playground namespace which match all labels specified in matchLabels. The NetworkPolicy restricts both ingress and egress traffic. For egress, the Pods are only allowed to speak DNS. For ingress, the Pods are only allowed to receive HTTP traffic, for example, from an Ingress Controller that handles TLS termination.

As you can see, this is a rather restrictive NetworkPolicy.

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: myapp-ck8s-user-demo
  namespace: cklein-playground
spec:
  podSelector:
    matchLabels:
      app.kubernetes.io/instance: myapp
      app.kubernetes.io/name: ck8s-user-demo
  policyTypes:
  - Ingress
  - Egress
  egress:
  - ports:
    - port: 53
      protocol: UDP
  ingress:
  - ports:
    - port: http
      protocol: TCP
```

Implementation-wise, a Kubernetes Container Network Interface (CNI) provider will watch both NetworkPolicies and Pods. Every time a Pod or NetworkPolicies change, the CNI will look up Pod IP addresses and configure the underlying kernel mechanism – often, but not always iptables – as needed.

Why are restrictive NetworkPolicies important?

In 2021, Log4Shell caught the world by surprise. Log4Shell was a zero-day vulnerability which affected almost all Java applications. Rated "maximum severity", Log4Shell allowed arbitrary remote code execution. Software engineers worldwide panicked and everyone rushed to patch their applications.

Log4Shell existed since 2013 and was only discovered 8 years later, making it a perfect example of a "known unknown". We know that such undiscovered vulnerabilities exist, but we don't know where. It is scary to think that some malicious actors might have known about Log4Shell years ahead of its public disclosure. Imagine what breaches they could have caused!

How can you protect yourself against the next zero-day vulnerability?

In order to exploit Log4Shell, your Java application had to connect to an attacker-controlled server. A restrictive NetworkPolicy would have prevented that, making a vulnerable application non-exploitable.

This is not an isolated case. In Dec 2022, someone uploaded a malicious Python package to trick developers into including it in their application. The library had exfiltration capabilities, which – you guessed it – needed the Pod to talk to an attacker-controlled IP address.

How to write the restrictive NetworkPolicies?

NetworkPolicies are fairly easy to write, once you know what traffic you want to allow to your Pods. However, the tricky part is knowing what traffic is expected.

In our experience, you cannot completely automate the process of blocking unexpected network traffic. There are tools to get insight into the network traffic of an application, such as service meshes, distributed tracing and even tcpdump. These tools help show the traffic produced by a Pod with a given usage and configuration. However, the shown traffic may be both "too much" and "too little". Too much, for example, when an anonymous usage reporting feature in a library is enabled by default. Too little, for example, when an external

system has many or unstable endpoints, and a network prefix is better than a single IP address.

At the end of the day writing the restrictive NetworkPolicies boils down to really understanding the application, what it's supposed to do and what it's not supposed to do. Setting the scope – as discussed in our previous guide – and ensuring each part of the tech stack is owned by an engineering team is a prerequisite.

To write the right NetworkPolicies, we proceeded as follows. We set up an environment where we could develop NetworkPolicies in a tight "build-measure-learn" loop, as the lean manifesto goes. Build (i.e., write) NetworkPolicies, measure the amount of traffic it blocks, learn more about the application and its expected traffic. We started with a "deny all" NetworkPolicy and added allow rules as required.

Fostering the right alerting culture is also needed: Without proof that the platform and application is healthy, each team has no choice but to act conservatively and configure permissive NetworkPolicies "just to make sure".

Final Thoughts

We often notice engineers forgetting to write NetworkPolicies. Therefore, best practice is to either block all Pod traffic by default, or enforce a safeguard to ensure NetworkPolicies exist. This helps "get the dumb stuff right", but is no cure for a poor security culture. After all, it is all too easy to write an "allow all" NetworkPolicy.

Restrictive NetworkPolicies are no excuse for leaving vulnerabilities unpatched. However, using them wisely helps you sleep better at night. Indeed, restrictive NetworkPolicies make some vulnerabilities harder to exploit. This gives you extra time to discover vulnerabilities, wait for a patch to be released and apply the patch to your Kubernetes environments. It allows you to enforce orderly vulnerability management, as opposed to panicking and making things worse, e.g., by causing downtime or data loss.

In the next section, we'll describe our philosophy around vulnerability management.

Further Reading

- ISO 27001 A13.1 Communications Security
- NetworkPolicies
- How to enforce NetworkPolicies

Vulnerability Management

You remember that time you forgot to lock your bike, but it did get stolen? That was a vulnerability. If you keep forgetting to lock your bike, a thief will eventually steal it.

This section will talk about vulnerability management: how to reason about vulnerabilities, how to discover them and how to act on them.

What are vulnerabilities?

Vulnerabilities are weaknesses in code. They can also be called “security bugs". Specifically, code is doing something it’s not supposed to do and that misbehavior puts data at risk. For example, in 2019 Kubernetes announced a container escape vulnerability. Needless to say, escaping from a container is a very powerful vector to exfiltrate data.

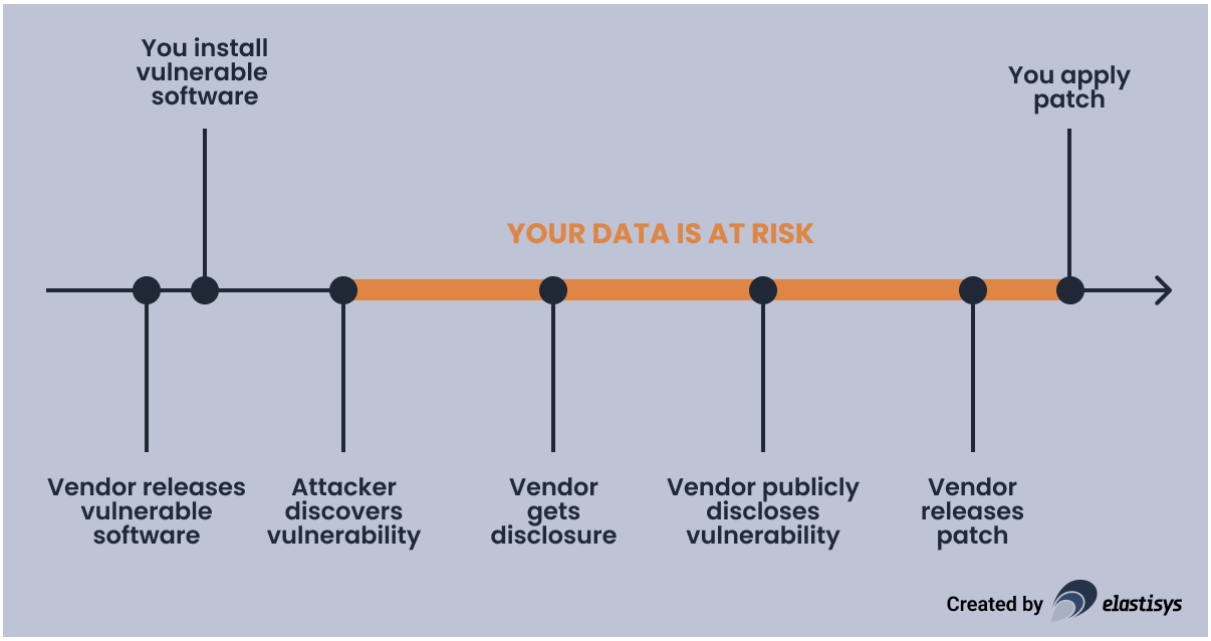
Note, however, that a vulnerability is not immediately a threat. A number of conditions need to be met:

1. Your application needs to depend on a software component which is vulnerable.
2. Your application needs to depend on the specific version which is vulnerable.
3. Your application needs to use vulnerable functionality.
4. Your application needs to use the vulnerable functionality in a way that the attacker can control it sufficiently. For example, attackers can easily control parameters in API calls. Your application needs to pass these parameters to the vulnerable functionality. Proper input validation and sanitization make this very hard.
5. The environment around your application allows the vulnerability to be exploited. We discussed this in the previous section.

As you can see, there is a long way from discovering a vulnerability to actually compromising data security. Still, just like with the bike lock metaphor above, vulnerabilities should not be left unattended. Hope is not a strategy. Also, your customers would rather hear that their data is protected thanks to process and not luck.

Vulnerabilities get a score or a criticality – i.e., critical, high, medium low. This is based on a number of factors and is meant to give an indication of how likely it is for the vulnerability to be exploitable, given 4 and 5. For example, a vulnerability which can be exploited remotely without privilege would get a higher score, then a vulnerability which requires privileged (sudo) access to the Kubernetes Node itself.

Vulnerability management generally refers to eliminating condition 2 above. Specifically, making a minimal software upgrade – usually called patching – to a non-vulnerable version.



Vulnerability Timeline

Before we discuss how you find out about a vulnerability that affects you, let us review the timeline of a vulnerability. A vulnerability gets born when a vendor – e.g., the authors of a software project – release a version containing that vulnerability. Of course, at that time, both the vendor and the general public were ignorant of the existence of the vulnerability. Sometime later, you will install that vulnerability.

In the unfortunate case, it is attackers who first discover the vulnerability. They would keep the vulnerability secret. They can then either sell it on the black market or use it directly for their malicious purposes. Eventually, some attackers get their hands on an exploit, which is a piece of software which can leverage the vulnerability to compromise data. This is when the "clock starts ticking" and your data is at risk.

In the fortunate case, it is ethical hackers – also called security researchers – who first discover the vulnerability. They would then privately disclose the vulnerability to the vendor, who would assess the risk and start working on a patch. At some point later, the vulnerability is publicly disclosed, hopefully at the same time as the patch.

Increasingly often, vulnerabilities are discovered "in the wild". This means that the vulnerability is actively exploited by attackers before the vendor learns about the vulnerability. These are called zero-day attacks. Eventually, someone's intrusion detection

system raises an alarm. Many security research hours later, the vulnerability becomes publicly known and is disclosed.

To sum up, it would be preferable for everyone to apply the patch before an exploit is produced. However, that is not always an option.

How do you find out about a vulnerability?

A vulnerability is publicly disclosed via several channels:

- Project-specific security mailing lists, such as kubernetes-security-announce.
- CVE® List – CVE stands for Common Vulnerabilities and Exposures, and has become the de facto hub for publishing vulnerabilities.
- Distribution-specific security trackers, such as the Debian Security Tracker.

Eventually, these vulnerabilities make it into the database of a container image vulnerability scanner, such as Trivy. Such scanners are advertised as making vulnerabilities more relevant, by carefully matching the vulnerability database with the software components you use in production (see conditions 1 and 2 above).

Unfortunately, container vulnerability scanning is a bit of a “crying wolf”. My hypothesis is that this is due to how scanners are assessed: There is a tendency to see more vulnerabilities as a “better” scanner. The psychology behind it is easy to understand: one prefers to have more information than too little. However, this leads to vulnerability reports being less actionable than my idealistic self would wish.

Hence, we nowadays recommend to use several signals as part of vulnerability management:

- Subscribe to the most important security announcements (e.g., Kubernetes, Grafana, Linux) and assess each vulnerability by hand.
- Use a container vulnerability scanner to scan container images before they go into production. For example, you may use the Harbor container registry which integrates with the Trivy container image scanner. You may even configure Harbor to block images from being pulled if they have vulnerabilities above a certain criticality.
- Use a container vulnerability scanner within the Kubernetes cluster. This ensures that vulnerabilities which are disclosed after a container image went into production doesn’t go unnoticed. The Trivy Operator does exactly that.

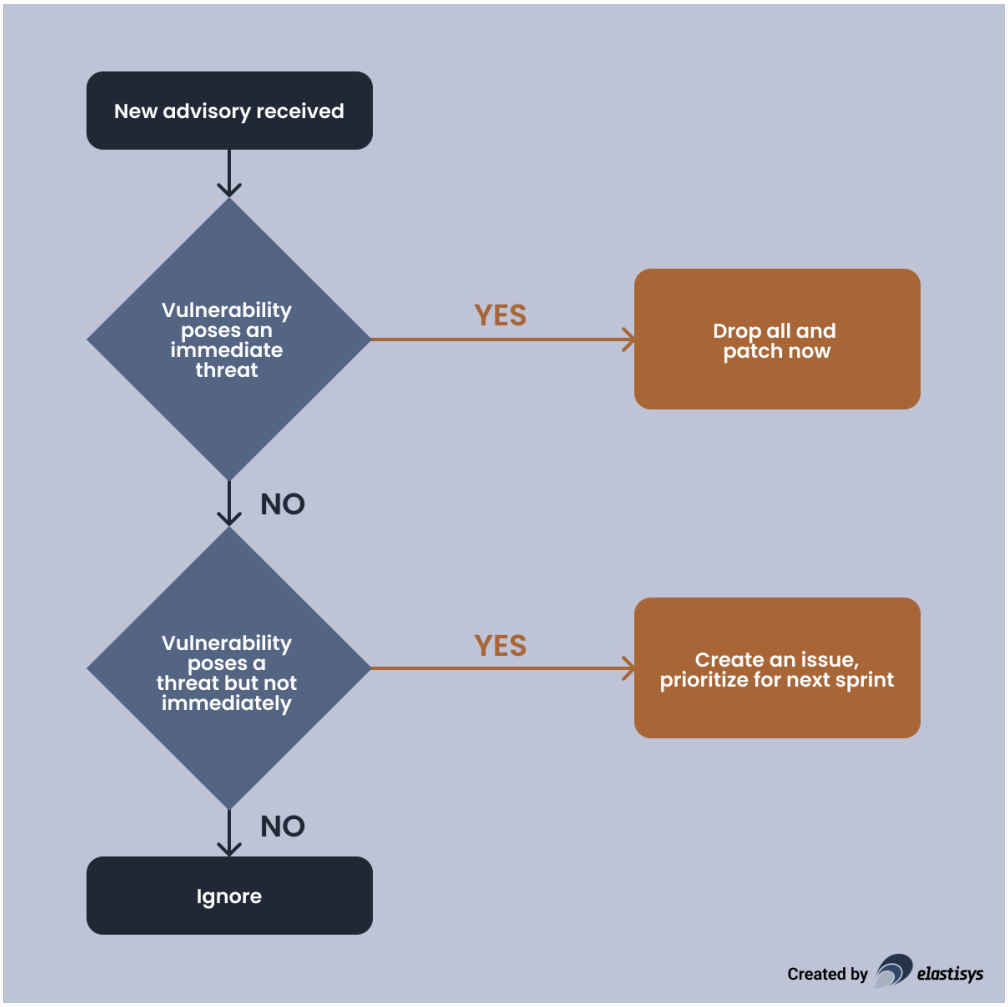
To reduce noise, use CVE allowlisting in Harbor or ignoreFile in the Trivy Operator to make reported CVEs more relevant. However, make sure not to overdo allowlisting CVEs.

How to act when you learn about a vulnerability?

Your goal should be to strike a balance between not missing out on important vulnerabilities and not going crazy due to the container scanner's "crying wolf" attitude. Therefore, we recommend the following approach.

The on-call person assesses each advisory and takes a decision as soon as possible. The decision is either:

- **Drop all and patch now**, if the vulnerability poses an immediate threat.
- **Create an issue to be prioritized for the next sprint**, if the vulnerability does pose a threat, but not an immediate one.
- **Ignore**, if the vulnerability does not pose a threat



We track the number of vulnerabilities of each Kubernetes platform release at the time when the release was produced. This is part of our QA process. The ambition is for the number of vulnerabilities not to go out-of-control. If it does, we create an investigation task for the next sprint, to make sure we are not missing any patches.

When looking at vulnerabilities, we prioritize as follows:

- Pods with critical vulnerabilities that are also publicly exposed via ingress and has higher privileges (root, host network access, host disk access, high kubernetes privileges, etc);
- Pods with critical vulnerabilities that are also publicly exposed via ingress;
- Pods with critical vulnerabilities that also has higher privileges (root, host network access, host disk access, high kubernetes privileges, etc.);
- Pods with critical vulnerabilities.
- Pods with high vulnerabilities in the same order as above.

In brief, we recommend container scanners like Trivy for a “sanity check” and security advisories for quick reaction to critical vulnerabilities.

Final Thoughts

We saw too many teams who were uncomfortable applying patches. Instead of thinking “we should really apply this patch to improve our security posture”, they were thinking “will we risk downtime”. This is not okay! Make sure to first foster maintenance habits, before tackling vulnerability management, as we described in our previous free guide.

Consider enforcing a trusted registry, for example via Gatekeeper/OPA, to make sure all images pass through vulnerability scanning.

Further Reading

- Trivy
- Trivy Operator
- Harbor: Vulnerability Scanning
- Harbor: Configure a Per-Project CVE Whitelist
- Enforce trusted registry

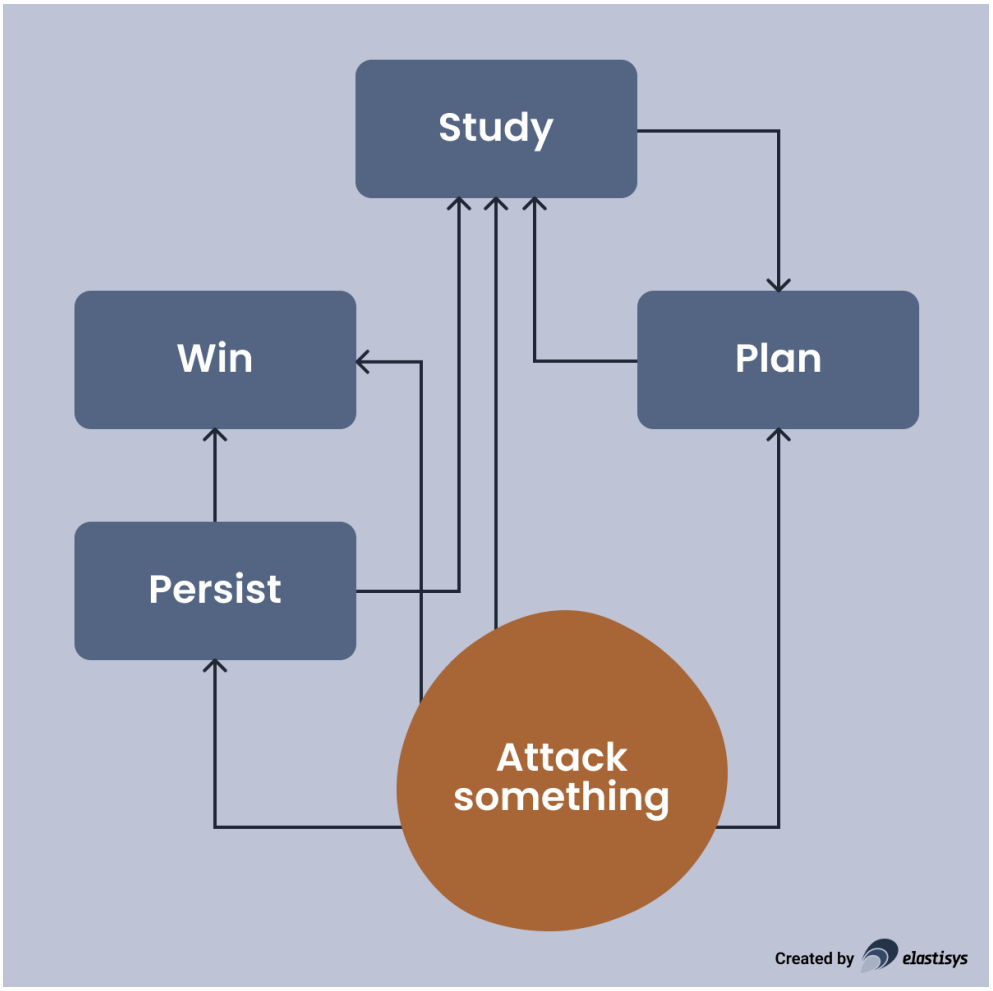
Intrusion Detection

As you read above, even for companies featuring the best vulnerability management, there is still an uncomfortably long window of opportunity for attackers to exploit zero-day vulnerabilities. So how do you deal with these “unknown unknowns”. Let’s dive into intrusion detection.

How does an attack work?

Before we dive into intrusion detection, let us learn more about intrusions. Let’s say an attacker found a vulnerability in your application or platform. They even found a way to exploit that vulnerability and make their way into the platform. Now what?

While it’s difficult to precisely lay out the attacker’s next steps, a general process would look as follows:



One of the first things an attacker – or a program on their behalf – would need to do is study their target. Note that, an attacker might not even know if they entered a VM or a container. Hence, they would issue commands, such as:

```
id
uname -a
cat /etc/lsb-release
ps -ef
df -h
netstat -nl
ifconfig
```

This will allows them to understand:

- Are they in a container or a virtual machine?
- Are they running as root or non-root?
- What kind of container are they running inside?

Once the initial study is over, they will try to download some tools and work towards their goal. Their goal could include:

- running Bitcoin mining;
- exfiltrating data;
- doing havoc to request a ransom.

Depending on the exact tool they are using, they will produce suspicious activity, such as:

- A container initiates an SSH connection, when it doesn't need such access.
- A container accesses files in suspicious locations, e.g., /etc.
- A container installs packages and executables, which it usually doesn't do.
- A container starts processes, which it usually doesn't do.

Note that, thanks to other security measures, many of these activities will fail. For example, thanks to blocking outbound network traffic, as described in the previous section, the attacker won't succeed in initiating an outbound SSH connection. However, an attacker will persist. Hence, picking up on these signals early will allow you to take countermeasures and stop the attacker from reaching their goal.

What is intrusion detection? How does it work?

By now you must have figured it out: Intrusion detection is a method for detecting suspicious activities generated by an ongoing attack. They serve as early warnings – data will be at risk – before the attacker makes irreparable damage.

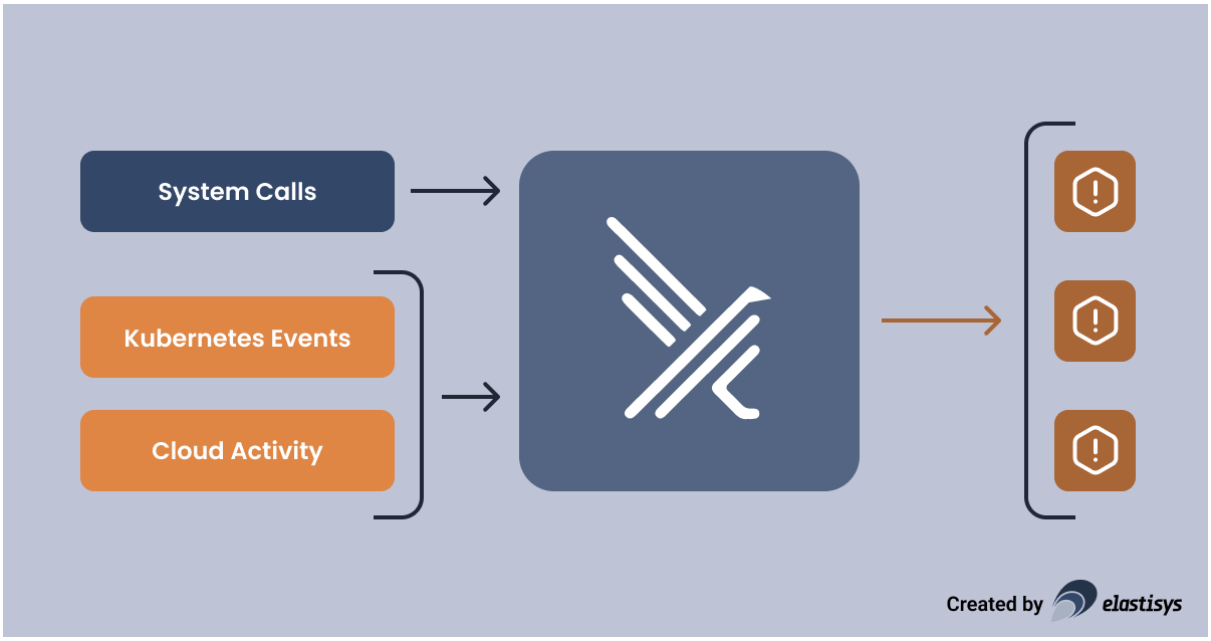
The most popular intrusion detection solution used in Kubernetes clusters is Falco. It is a Cloud Native Foundation (CNCF) incubated project, which means that:

- It is open source (Apache 2.0), hence no licensing costs.
- It is open governance, so no single vendor can derail the project in their interest.
- It is likely to stay popular among users for the foreseeable future.
- It is likely to get significant contributions, in particular features and security patches.

To understand how Falco works, you need to understand what system calls (syscalls) and eBPF are.

Syscalls is the API that the Linux kernel exposes to applications. For example, an application establishing an SSH connection needs to call the connect syscall. Similarly, an application wanting to read a file will first have to call the open syscall. An application wanting to launch a new process will have to use a combination of clone and execve. As you might notice, one can get a pretty good understanding of what an application is up to when looking at what syscalls it invokes. You can get an even better understanding by looking at the syscall input parameters. So how can we observe what syscalls an application invokes?

eBPF is a technology to achieve exactly that. Without going into details, eBPF allows an intrusion detection system, such as Falco, to set up probes. Each time the application invokes a syscall, Falco gets an event. These events are then matched against a rule engine to determine if the activity is suspicious or not.



Rules: The Key to Relevant Intrusion Detection

As you might expect, Falco sees quite a lot of events. The key to relevant intrusion detection is having good rules in place.

Rules can be found at three levels:

1. Rules distributed with Falco: These are relevant for a wide range of Kubernetes platforms.
2. Rules distributed with the Kubernetes distribution: For example, Elastisys Compliant Kubernetes uses a few images which need special rules with Falco.
3. Environment-specific rules: These rules are only relevant for a specific environment. For example, in one of our environments the application performs FTP uploads, which is marked as suspicious by the rules distributed with Falco and Elastisys Compliant Kubernetes.

A healthy alerting culture, as described in our previous guide, is a solid foundation for intrusion detection. The team needs to feel empowered to tune the rules to the needs of the individual application and environment. Otherwise, Falco will end up “crying wolf” and you won’t get the benefits of intrusion detection.

We recommend the following process:

1. Start with the rules distributed by Falco or your Kubernetes distribution. These are likely to overalert. Configure your On-call Management Tool (OMT) to classify these

alerts as P4, i.e., review regularly. At this point, they should not steal the attention of your team, but should be stored for later review.

2. After a few weeks, collect all Falco alerts in a sheet. Create a matrix where the rows are the environments and the columns the Falco rule name. Note down how often each alert occurred for each environment, as well as the total.

3. Start with the alerts that are most frequent and investigate each of these alerts by looking at the exact Falco alert line. Such an alert might look as shown below.

4. It is rather difficult to decide if an alert is relevant or not and what rule to fix. Consider discussing these alerts in all-hands engineering meetings. For example, we investigated the alert below and deduced the following:

    a. This alert is caused by PostgreSQL managed by the Zalando Operator.

    b. The behavior is normal for the way the Zalando Operator implements Point-in-Time Recovery via WAL-E.

    c. The alert is caused by Falco expecting a different proc.cmdline than the default rule.

5. Update the rule for the environment.

6. If the rule is non-environment-specific, consider upstreaming the rule either to your Kubernetes distribution or the Falco project. For example, for the alert below, we determined that the updated rule is relevant for the whole Falco community. Hence, we upstreamed it. Once the rule is upstreamed and you update Falco, the rule can be removed from the environment-specific configuration.

7. Repeat steps 2-6 until you found the right level of alerting.

```
{
 "priority":"Notice",
 "rule":"DB program spawned process",
 "time":"2022-03-09T07:30:16.755996921Z",
 "output_fields":{
   "container.id":"7b2808fc84dc",
   "container.image.repository":
     "registry.opensource.zalan.do/acid/spilo-14",
   "evt.time":1646811016755996921,
   "k8s.ns.name":"postgres-system",
   "k8s.pod.name":"elastisys-production-13-a-1",
   "proc.cmdline":"sh -c envdir \"/run/etc/wal-e.d/env\" wal-g wal-push
\"pg_wal/000000090000011000000EC\"",
   "proc.pname":"postgres",
   "user.loginuid":-1,
   "user.name":null
 }
}
```


Eventually, you should reach the right level of alerting, so that Falco alerts can be classified as P1, i.e., “2am alerts” or “need attention immediately”. This will ensure that you catch even attacks trying to stay undetected by starting on a Friday evening.

Falco’s rule engine is extremely powerful and we never found it to “get in our way”. Unfortunately, coming up with the right rules requires expert knowledge. However, by contributing to Falco, you will use your experts as efficiently as possible.

Final Thoughts

Depending on your security goals, you might also find it useful to let Falco monitor the Kubernetes audit log. For example, this helps you alert on exec/attach against your production environments which – depending on your workflow – is extremely suspicious.

Further Reading

- KubeCon NA 2019 CTF
- The Falco Project

- Falco: Default Rules
- Falco: Kubernetes Audit Rules
- Elastisys Compliant Kubernetes Falco Rules

Takeaways

In our previous guide we set up basic “Kubernetes security hygiene”. This guide helps you further improve the security of your Kubernetes platform:

- **Block unnecessary network traffic** to make vulnerabilities harder to exploit and make it harder for attackers to move laterally.
- **Systematically apply security patches** to make sure vulnerabilities don’t pile up and eventually get exploited by attackers.
- **Use intrusion detection** and configure it properly to get early warnings of a potential attack and defend against zero-day vulnerabilities.

With the right NetworkPolicies, the right vulnerability management and the right intrusion detection rules, you should be well equipped to maintain the stability and security of your Kubernetes platform against all "known unknowns" and "unknown unknowns".

Elastisys AB • elastisys.com