Monitoring Keamanan *Runtime* pada Kubernetes Menggunakan Falco

Ryan Fadhillah[1], Nur Rohman Rosyid[1,*]
[1]Departemen Teknik Elektro dan Informatika, Sekolah Vokasi, Universitas Gadjah Mada;
ryanfadhillah@mail.ugm.ac.id
[*]Korespondensi: nrohmanr@ugm.ac.id;

Abstract – The use of Kubernetes as a container management platform has significantly increased in recent years. With the growth of Kubernetes usage, security has become a crucial issue that must be taken seriously. Threat detection during the runtime phase is the most important security capability that Kubernetes should possess. This is because runtime security serves as the last line of defense in a security system. To address this issue, this research focuses on developing a runtime security monitoring tool using Falco. Additionally, this project develops methods for tuning rules to enhance the detection capabilities of Falco's default rules. The detection capability testing process is conducted against three attack scenarios included in the OWASP Top 10 Cloud-Native Application Security risks, including Remote Code Execution (RCE), exfiltration using common Linux binaries, and privileged container. By tuning several rules according to the testing scenarios, this project successfully improves the detection effectiveness of malicious activities in a Kubernetes environment. This significantly contributes to organizations that aim to implement Falco for securing the runtime phase in their Kubernetes infrastructure.

Keywords – Kubernetes, Runtime Security, Falco, Rule Tuning, OWASP Top 10 Cloud-Native Application Security

Intisari – Penggunaan Kubernetes sebagai platform manajemen kontainer telah semakin meluas dalam beberapa tahun terakhir. Seiring dengan pertumbuhan penggunaan Kubernetes, keamanan menjadi isu krusial yang perlu diperhatikan secara serius. Deteksi ancaman pada fase *runtime* merupakan kapabilitas keamanan Kubernetes paling penting untuk dimiliki. Hal ini dikarenakan keamanan pada fase *runtime* merupakan garis pertahanan terakhir dalam sebuah sistem keamanan. Untuk mengatasi masalah tersebut, penelitian ini berfokus pada pengembangan alat monitoring keamanan *runtime* menggunakan Falco. Selain itu, penelitian ini juga mengembangkan metode untuk melakukan *tuning rules* untuk meningkatkan kapabilitas deteksi dari *rule default* Falco. Proses pengujian kapabilitas deteksi dilakukan terhadap tiga skenario serangan yang termasuk kedalam daftar resiko yang ada pada OWASP Top 10 *Cloud-Native Application Security*, di antaranya yaitu *Remote Code Execution* (RCE), *exfiltration using common Linux binaries,* dan *privileged container*. Melalui proses *tuning* terhadap beberapa *rule* sesuai dengan skenario pengujian, penelitian ini berhasil meningkatkan efektivitas deteksi terhadap *malicious activity* pada lingkungan Kubernetes. Hal ini memberikan kontribusi signifikan bagi organisasi yang ingin mengimplementasikan Falco dalam mengamankan fase *runtime* pada infrastruktur Kubernetes.

Kata kunci – Kubernetes, *Runtime Security*, Falco, *Tuning rule*, OWASP Top 10 *Cloud-Native Application Security*

I. PENDAHULUAN

Penggunaan Kubernetes sebagai platform manajemen kontainer telah semakin meluas dalam beberapa tahun terakhir. Kubernetes adalah salah satu bukti nyata masifnya perkembangan teknologi komputasi awan, virtualisasi, dan otomasi jaringan. Berdasarkan data dari Cloud Native Computing Foundation, Kubernetes mendapatkan predikat sebagai salah satu proyek teratas di Github yang diukur dengan jumlah bintang yang diberikan oleh pengguna Github, dengan dokumentasi yang sangat baik dan memiliki komunitas yang besar [1]. Hal ini dikarenakan Kubernetes memberikan fleksibilitas dan skalabilitas yang tinggi dalam pengelolaan aplikasi berbasis kontainer.

Kubernetes berasal dari pengalaman Google dengan mengelola sistem terdistribusi yang luas melalui sistem manajemen klaster internal yang dikenal sebagai Borg, yang menjalankan ribuan aplikasi di ribuan komputer. Kubernetes mengadopsi prinsip Borg dan mewarisi keunggulan skalabilitas, ketersediaan tinggi, dan efisiensi, yang menjadikannya platform *open-source* yang dapat diandalkan untuk pengelolaan aplikasi modern [1].

Namun, seiring dengan pertumbuhan penggunaan Kubernetes, keamanan menjadi isu krusial yang perlu diperhatikan secara serius. Sebuah studi terbaru, mengungkapkan bahwa *human-error* merupakan penyebab utama dalam 95% *data breach* yang terjadi di dunia [2]. Dalam lingkungan yang semakin kompleks dan terdistribusi, menjaga keamanan Kubernetes menjadi semakin menantang karena adanya potensi serangan dan ancaman yang dapat muncul dari berbagai sumber. Ditambah lagi, konfigurasi pada Kubernetes memang tidak didesain *secure* secara *default* [1]. Penelitian [3] mengatakan bahwa miskonfigurasi merupakan penyebab utama dari *security incident* yang terjadi di Kubernetes dengan persentase 53% responden mengalaminya.

Ketika terdapat miskonfigurasi yang menyebabkan adanya *security incident*, *indicator of attack* akan selalu muncul untuk menandakan terjadinya suatu *security incident*. Semua *indicator of attack* akan tercermin pada waktu *runtime* dikarenakan *runtime security* berperan sebagai garis pertahanan terakhir dalam sistem keamanan [4]. Berdasarkan NIST.SP-800-190 *framework*, penggunaan alat pertahanan berbasis *runtime* yang sadar akan kontainer menjadi sangat penting untuk dapat mencegah, mendeteksi, dan merespons

ancaman yang ada pada kontainer selama *runtime*. Maka dari itu, pemantauan keamanan *runtime* menjadi sangat penting dalam mendeteksi serangan secara *real-time* dan merespons ancaman dengan cepat.

Berdasarkan penelitian yang dilakukan oleh [3] dinyatakan bahwa Falco merupakan satu-satunya alat *open-source* yang paling banyak digunakan untuk melakukan monitoring terhadap kemanan *runtime* pada Kubernetes. Referensi [5] juga menyatakan bahwa Falco dapat melakukan monitoring keamanan *runtime* berbasis aturan untuk mendeteksi berbagai potensi *malicious activity* dari beban kerja di lingkungan terkontainerisasi.

Untuk mengatasi berbagai permasalahan di atas, penelitian ini menitikberatkan pada pengembangan sistem deteksi keamanan *runtime* pada Kubernetes untuk mendeteksi *malicious activity* dengan memanfaatkan perangkat lunak *open source* bernama Falco. Falco secara *default* menggunakan peristiwa *syscall* untuk mengidentifikasi pola aktivitas berbahaya. Namun, untuk meningkatkan efektivitas deteksi, administrator perlu mengatur aturan deteksi dalam format YAML yang sesuai dengan kasus penggunaan tertentu. Meskipun Falco dilengkapi dengan aturan bawaan, beberapa di antaranya masih memerlukan penyesuaian tambahan untuk keperluan spesifik. Maka dari itu, penilitian ini juga akan mengusulkan metode untuk melakukan *tuning rules* atau melakukan peningkatan kapabilitas *rules* terhadap tiga skenario serangan yang akan diujikan. Pemilihan skenario serangan didasarkan pada serangan-serangan yang termasuk kedalam daftar resiko yang ada pada OWASP *Top 10 Cloud-Native Application Security*. Skenario serangan yang akan diuji dalam penelitian ini meliputi *Remote Code Execution* (RCE), *exfiltration using common Linux binaries* dan *deployment of privileged container*.

II. DASAR TEORI

A. Kubernetes dan *Containerization*

Kubernetes merupakan alat orkestrasi kontainer yang digunakan untuk melakukan manajemen aplikasi yang dikontainerisasi. Kontainerisasi merujuk pada proses pengemasan aplikasi beserta semua komponen yang diperlukan seperti file konfigurasi terkait, *library*, dan dependensi yang diperlukan untuk memastikan aplikasi dapat beroperasi secara efisien [6].

Perusahaan memiliki beragam opsi platform yang tersedia untuk menggunakan Kubernetes. Jika anggaran perusahaan terbatas, pilihan yang tepat adalah menggunakan platform Kubernetes *self-manage* yang *open source* seperti Minikube, K3s, Kind, MicroK8s, dan lain sebagainya. Namun, hal ini mungkin menimbulkan tantangan dalam pengelolaan dan pemeliharaan sehingga mengharuskan perusahaan untuk memperhatikan berbagai aspek teknis. Sebaliknya, jika perusahaan memiliki anggaran lebih besar, disarankan untuk mempertimbangkan penggunaan *managed* Kubernetes (Kubernetes *as a service*) seperti Google GKE (Google Kubernetes Engine), Amazon EKS (Elastic Kubernetes Service), Microsoft AKS (Azure Kubernetes Service), Openshift Kubernetes Engine, Digital Ocean Kubernetes (DOKS), dan lain sebagainya. Dengan menggunakan layanan ini, perusahaan tidak perlu repot dalam mengurus atau mengelola Kubernetes secara mandiri, sehingga dapat fokus pada pengembangan aplikasi dan peningkatan produktivitas [7].

B. *Container Runtime Security*

Keamanan *runtime* kontainer bukan hanya aspek tambahan, namun merupakan elemen fundamental dan langkah proaktif untuk menjaga keamanan dan keandalan aplikasi yang terkontainerisasi selama fase *runtime*-nya. Lingkungan terkontainerisasi ini dirancang untuk ringan, portabel, dan dapat diskalakan. Namun, mereka juga membuka potensi kerentanan keamanan baru selama *runtime*. Oleh karena itu, diperlukan langkah-langkah khusus untuk memastikan keamanannya selama fase *runtime* [8], [9].

Kontainer memiliki siklus hidup yang jelas, di antaranya yaitu *build, ship*, dan *run*. Meskipun telah diberikan penekanan yang signifikan pada keamanan selama tahap *build* (misalnya, *container image scanning*) dan tahap *ship* (misalnya, orkestrasi kontainer yang aman), tahap *runtime* sering kali kurang diperiksa. Padahal, tahap ini sangat penting karena pada saat itu aplikasi berjalan dan memproses data secara langsung, menjadikannya target yang menarik bagi penyerang potensial.

Menurut penelitian [3], *runtime threat detection/response* dianggap sebagai kapabilitas yang sangat penting untuk diterapkan dalam lingkungan Kubernetes. Sebanyak 69% responden mengidentifikasikannya sebagai kapabilitas keamanan yang harus dimiliki, diikuti dengan *configuration management* dan *image scanning/vulnerability management* masing-masing dengan persentase 68% dan 65%.

C. *System Call*

Dalam komputasi, *system call* atau yang sering disingkat *syscall* adalah cara terprogram di mana suatu program komputer meminta layanan dari kernel sistem operasi di mana program tersebut dijalankan. Sebuah program komputer akan melakukan *system call* ketika melakukan permintaan kepada kernel sistem operasi. *System call* adalah satu-satunya titik masuk ke sistem kernel [10]. Oleh karena itu, dengan memantau *syscall* pada Linux Kernel yang dibuat oleh proses terkontainerisasi, semua *malicious activity* dapat dideteksi ataupun diblok.

Cara kerja *syscall* erat hubungannya dengan *user mode* dan *kernel mode*. *User mode* merupakan mode terbatas yang membatasi akses ke sumber daya sistem, sementara *kernel mode* merupakan mode yang memiliki *privileged* yang memungkinkan akses ke sumber daya system. Aplikasi yang berada di *user mode* harus membuat *system call* untuk dapat mengakses sumber daya yang ada di *kernel mode* atau melakukan operasi yang memiliki *privileged* tinggi [10].

D. eBPF

eBPF (*extended Berkeley Packet Filter*) merupakan teknologi canggih yang berasal dari kernel Linux, memungkinkan eksekusi program terisolasi dalam konteks kernel tanpa perlu memodifikasi kode sumber atau memuat modul tambahan. Teknologi ini merevolusi cara kerja sistem operasi dalam memperluas fungsi observabilitas, keamanan, dan jaringan secara dinamis dan efisien. Dengan eBPF, pengembang dapat menambahkan fitur ke sistem operasi saat runtime, yang sebelumnya sulit dilakukan karena sifat kernel yang stabil dan sensitif terhadap perubahan.

Saat ini, eBPF telah digunakan dalam beragam aplikasi seperti *load balancing* dan *networking* berkinerja tinggi di lingkungan *cloud-native*, penguatan keamanan aplikasi, dan kontainer, serta observabilitas sistem dengan *overhead* rendah. Salah satu implementasi nyatanya adalah dalam proyek *open-source* Falco, sebuah *tool* untuk deteksi ancaman *runtime*. Melalui integrasi eBPF, Falco dapat menganalisis *system call* secara *real-time* dengan lebih efisien dan aman, terutama dalam lingkungan modern seperti Kubernetes, di mana pendekatan tradisional seperti *kernel probe* memiliki keterbatasan. Kombinasi ini memperluas kapabilitas deteksi dan respons keamanan yang adaptif di ekosistem *cloud* masa kini [11].

E. Falco

Falco adalah proyek keamanan *runtime* pada *cloud native* pertama yang bergabung dengan CNCF (Cloud Native Computing Foundation) sebagai proyek dengan *maturity graduated. Maturity graduated* merupakan *maturity* tertinggi yang diberikan kepada proyek yang masuk kedalam CNCF. Jika suatu proyek telah berada di tingkat ini, itu menunjukkan bahwa proyek tersebut sudah siap untuk diimplementasikan di lingkungan *production*, memiliki komunitas yang aktif serta komitmen terhadap keberlanjutan proyek tersebut [12]. Falco dirancang untuk mendeteksi keamanan *runtime* dan memberikan *alert* jika ada perilaku abnormal dan potensi ancaman keamanan secara *real-time*. Falco bertindak sebagai kamera keamanan yang terus mendeteksi perilaku tak terduga, perubahan konfigurasi, intrusi, dan pencurian data secara *real-time* [13].

F. *Mean Time to Detect* (MTTD)

Mean Time to Detect (MTTD) adalah ukuran yang mengacu pada waktu yang diperlukan dari saat masalah pertama kali muncul hingga saat ditemukan oleh orang atau sistem monitoring. Dalam dunia *cybersecurity*, MTTD dapat diartikan sebagai berapa rata-rata waktu yang dibutuhkan sejak sebuah *event* serangan siber terjadi hingga *event* tersebut di deteksi oleh sistem *monitoring*. KPI ini melacak seberapa efektif departemen IT dan organisasi dalam menghindari gangguan jangka panjang yang dapat timbul. Semakin cepat *event* di deteksi, semakin cepat pula tim keamanan internal dapat melakukan *incident handling* untuk meminimalisir dampak resiko yang bisa muncul akibat adanya serangan tersebut [14].

Rumus *Mean Time to Detect* (MTTD) dapat dilihat pada (1) [14]. Untuk menghitung MTTD, dapat menjumlahkan semua waktu deteksi tiap insiden, lalu membaginya dengan jumlah insiden. Misalnya, jika total waktu deteksi untuk bulan Januari adalah 850 menit dan terdapat 12 insiden yang dilaporkan, maka nilai *Mean Time to Detect* akan menjadi 850 menit dibagi 12 menjadi 70.83 menit. Dengan memantau MTTD secara teratur, organisasi dapat mengidentifikasi pola waktu deteksi dan mengevaluasi efektivitas sistem deteksi masalah untuk meningkatkan kinerja dan meminimalkan dampak negatif pada bisnis.

$$MTTD = \frac{\sum_{i=1}^{n}\left(t_{deteksi_i} - t_{insiden_1}\right)}{n} \quad (1)$$

dengan:

$t_{insiden_i}$ = waktu terjadinya insiden ke-i

$t_{deteksi_i}$ = waktu insiden ke-i terdeteksi

n = Jumlah total insiden yang dilaporkan

III. METODOLOGI

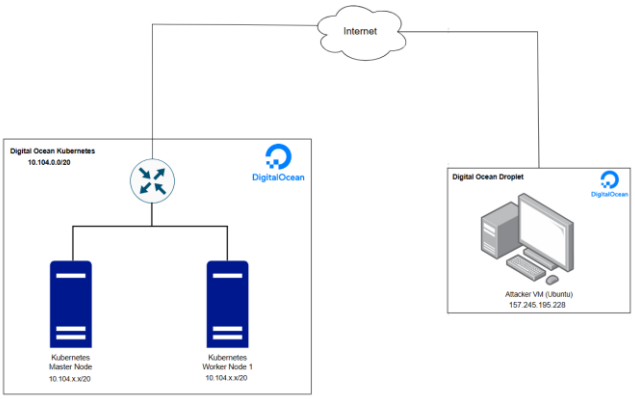
A. Lingkungan Eksperimen

Penelitian ini menggunakan *Cloud Service Provider (CSP)* dari Digital Ocean. Digital Ocean merupakan salah satu *cloud provider* yang memiliki banyak layanan. Penelitian ini menggunakan dua layanan dari Digital Ocean di antaranya yaitu Digital Ocean Droplets dan Digital Ocean Kubernetes. Tabel 1 akan menunjukkan spesifikasi tiap layanan yang digunakan dalam penelitian ini.

Tabel 1. Spesifikasi aset

Aset	Role	Spesifikasi
Digital Ocean Kubernetes	Klaster Kubernetes	8 vCPU, 8 GB RAM, 160 GB SSD
Digital Ocean Droplets	VM Attacker	1 vCPU, 2 GB RAM, 50 GB SSD

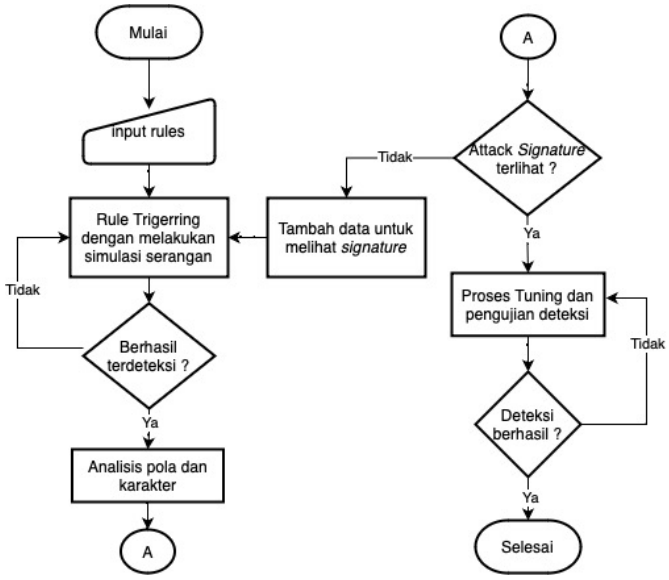
Topologi jaringan dari sistem yang digunakan pada penelitian ini dapat dilihat pada Gambar 1. Rancangan sistem yang digunakan pada penelitian ini menggunakan layanan Digital Ocean Kubernetes (DOKS) dengan satu *master node* dan satu *worker nodes*. Seluruh sistem Kubernetes memiliki alamat IP dengan subnet 10.104.0.0/20. Selain itu, penelitian ini juga menggunakan layanan Digital Ocean Droplets sebagai VM Penyerang yang akan melakukan skenario serangan dengan Alamat IP 157.245.195.228.



Gambar 1. Topologi jaringan sistem

B. Penyetelan *Rule*

Secara *default,* Falco telah memiliki beberapa *rule* default yang dapat langsung digunakan. Akan tetapi, beberapa di antaranya masih memerlukan proses *tuning* lebih lanjut untuk skenario yang lebih spesifik. Gambar 2 memperlihatkan proses *tuning* yang dilakukan.



Gambar 2. Proses *tuning rule*

Proses *tuning* Falco diawali dengan menyimulasikan berbagai skenario serangan menggunakan aturan *default* dari repositorinya. Langkah ini bertujuan mengevaluasi efektivitas deteksi awal serta mengidentifikasi aspek yang perlu disesuaikan untuk meningkatkan akurasi deteksi. Hasil awal dari langkah ini memberikan wawasan awal mengenai performa *rule* dalam mengenali aktivitas berbahaya dan menjadi dasar proses penyetelan selanjutnya.

Langkah selanjutnya adalah memeriksa pola atau *signature* khas yang terkait dengan setiap skenario yang di-*trigger* pada langkah pertama. Analisis digunakan untuk memahami detail khusus dari *malicious activity* yang terdeteksi, memberikan wawasan tentang ciri-ciri uniknya.

Jika tidak melihat karakteristik ataupun pola yang jelas dari output yang diberikan, maka perlu menambahkan data tambahan untuk membuat *output* lebih informatif. Penambahan ini bertujuan untuk memberikan pemahaman yang lebih jelas tentang perbedaan karakteristik di setiap pengujian skenario. Untuk daftar lengkap *event* yang tersedia pada dokumentasi Falco [15].

Dengan mempertimbangkan wawasan yang diperoleh dari analisis pola, langkah selanjutnya yaitu melakukan *tuning rule* Falco terhadap skenario tertentu. Proses *tuning rule* dilakukan dengan mempertimbangkan skenario spesifik dari sebuah ancaman. Dalam hal ini, perlu pemahaman mendalam dari sebuah skenario serangan agar memudahkan dalam proses *tuning rule*. Proses *tuning* ini bertujuan untuk meningkatkan kemampuan Falco dalam mendeteksi pola-pola unik yang terkait dengan berbagai ancaman.

Setelah melakukan proses *tuning rule*, aturan tersebut dapat disimpan pada direktori "/etc/falco". Secara *default,* terdapat dua file dalam direktori tersebut, di antaranya yaitu file "falco.yaml" untuk konfigurasi Falco dan "falco_rules.yaml" untuk aturan *default*. Jika ingin menambahkan aturan baru di luar aturan *default*, pengguna dapat membuat file baru dengan nama "falco_rules.local.yaml". Setelah itu, dapat mengulangi proses *tuning* seperti pada Gambar 2.

Pada penelitian ini, proses *tuning rule* berfokus pada tiga skenario yang diujikan, di antaranya yaitu *Remote Code Execution, exfiltration using common Linux binaries*, dan *deployment privileged container*. Berikut penjelasan dari tiap-tiap skenario.

1. Skenario *Remote Code Execution*

Untuk skenario serangan *Remote Code Execution* (RCE), penelitian ini menggunakan aturan yang berasal dari Repositori Falco di Github: "falco_rules.yaml". Secara *default*, aturan ini sudah diaktifkan ketika menggunakan Falco untuk pertama kali. Namun, selama pengujian, beberapa skenario serangan tidak terdeteksi oleh aturan ini, sehingga perlu dilakukan proses *tuning* lebih lanjut. Gambaran secara umum cara kerja *rule* RCE pada Falco dapat dilihat diagram YAML pada Gambar 3.

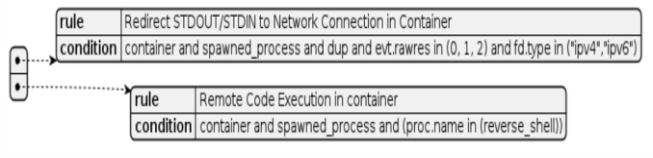
rule	Redirect STDOUT/STDIN to Network Connection in Container
condition	dup and container and evt.rawres in (0, 1, 2) and fd.type in ("ipv4", "ipv6")

Gambar 3. Diagram YAML *rule* sebelum *tuning* dengan RCE

Setelah dilakukan berbagai percobaan serangan dan melihat pola dari tiap-tiap serangan, langkah selanjutnya yaitu melakukan modifikasi atau penambahan kondisi pada aturan tersebut untuk meningkatkan kapabilitas aturan Falco sesuai dengan meningkatnya variasi serangan untuk skenario

Remote Code Execution (RCE). Proses *tuning* ini secara khusus terkait dengan penggunaan biner Linux umum yang sering digunakan dalam aktivitas *reverse shell*. Penelitian ini melakukan penyempurnaan aturan dengan menambahkan kondisi (*spawned_process and* (*proc.name in* (*reverse_shell*)) untuk mendeteksi penggunaan biner Linux umum, seperti *nc*, *ncat*, *socat*, *telnet*, *zsh*, dan *rushcat*, sebagai indikator potensial dari upaya *Remote Code Execution* (RCE). Penambahan kondisi ini dapat mengurangi adanya *false negative* dalam aturan ini. Selain itu, penelitian ini juga menambahkan kondisi pengecualian terhadap *proc.name in* ("kubelet", "dockerd") agar aturan ini tidak akan menandainya sebagai peringatan dikarenakan process tersebut merupakan process yang dilakukan oleh komponen internal di dalam kubernetes dan bukan merupakan proses yang berbahaya. Penambahan kondisi pengecualian ini ditujukan untuk mengurangi adanya *false positive* dalam aturan ini.

Gambaran secara umum terhadap cara kerja *rule* RCE pada Falco dapat dilihat diagram YAML pada Gambar 4.

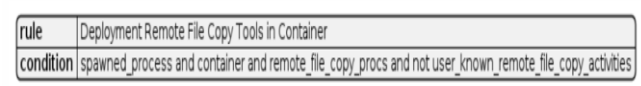


Gambar 4. Diagram YAML *rule* setelah *tuning* dengan RCE

2. Skenario *Exfiltration using Common Linux Binaries*

Untuk skenario serangan *exfiltration using common Linux binaries*, penelitian ini menggunakan aturan yang berasal dari repositori Falco di Github: "falco-incubating_rules.yaml", menandakan bahwa aturan tersebut diidentifikasi oleh para ahli sebagai aturan yang akan menangani skenario penggunaan terhadap serangan yang lebih spesifik, yang mungkin relevan untuk beberapa pengguna, tetapi tidak relevan terhadap pengguna lainnya. Secara *default*, aturan ini sudah diaktifkan ketika menggunakan Falco untuk pertama kali. Namun, aturan *default* hanya mendefinisikan empat biner Linux umum terkait eksfiltrasi, sehingga memerlukan modifikasi tambahan untuk meningkatkan kemampuan deteksi untuk skenario eksfiltrasi. Biner Linux yang masuk kedalam aturan *default* di antaranya *rsync*, *scp*, *sftp*, dan *dcp*.

Namun, selama pengujian, beberapa skenario serangan tidak terdeteksi oleh aturan ini, sehingga perlu dilakukan proses *tuning* lebih lanjut. Cara kerja *rule exfiltration using common Linux binaries* pada Falco sebelum proses *tuning* dapat dilihat diagram YAML pada Gambar 5.



Gambar 5. Diagram YAML *rule* sebelum *tuning* dengan *Exfiltration Using Common Linux Binaries*

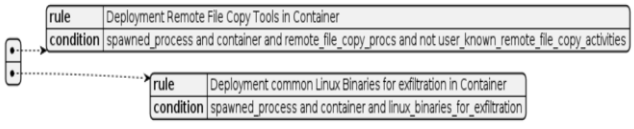
Setelah dilakukan berbagai percobaan serangan dan melihat pola dari tiap-tiap serangan, penelitian ini melakukan modifikasi atau penambahan kondisi pada aturan tersebut untuk meningkatkan kapabilitas aturan Falco sesuai dengan meningkatnya variasi serangan untuk skenario *exfiltration using common Linux binaries*. Proses *tuning* ini secara khusus terkait dengan penggunaan biner Linux umum yang sering digunakan dalam lingkungan Linux yang dapat disalahgunakan untuk melakukan eksfiltrasi, seperti *wget*, *whois*, *bash*, *openssl*, *curl*, dan *bash*. Penambahan kondisi pada aturan dilakukan untuk meningkatkan kapabilitas deteksi dari skenario *exfiltration using common Linux binaries* dapat dilihat pada Tabel 2.

Tabel 2. Penambahan kondisi pada *rule exfiltration using common Linux binaries*

Binari Linux	**Penambahan Kondisi**
wget	(proc.name = "wget" and (proc.cmdline contains "--post-file" or proc.cmdline contains "--post-data"))
whois	(proc.name = "whois" and (proc.cmdline contains "-h" or proc.cmdline contains "-p"))
bash	(proc.name = "bash" and proc.cmdline contains "-c")
curl	(proc.name = "curl" and (proc.cmdline contains "-x POST" or proc.cmdline contains "-d"))
KSH	(proc.name = "ksh" and proc.cmdline contains "-c")
openssl	(proc.name = "openssl" and (proc.cmdline contains "-connect" or proc.cmdline contains "s_client"))

Penambahan kondisi pada Tabel 2 dilakukan dengan mendeteksi *event* "proc.name" untuk mendeteksi binari Linux yang berjalan serta "proc.cmdline" untuk menentukan parameter apa saja pada binari linux tersebut yang dapat disalahgunakan untuk melakukan eksfiltrasi. Hal ini dilakukan karena beberapa dari binari Linux tersebut juga masih sering digunakan oleh *developer* internal, sehingga kondisi pada *rules* perlu disesuaikan untuk mengurangi kemungkinan terjadinya *false positive*.

Secara umum, cara kerja *rule exfiltration using common Linux binaries* pada Falco setelah proses *tuning* dapat dilihat diagram YAML pada Gambar 6.



Gambar 6. Diagram YAML *rule* setelah *tuning* dengan *exfiltration using Common Linux Binaries*

3. Skenario *Deployment of Privileged Container*

Untuk skenario *deployment of privileged container*, penelitian ini menggunakan aturan yang berasal dari repositori Falco di Github: "falco-incubating_rules.yaml" yang menandakan bahwa aturan tersebut diidentifikasi oleh para ahli sebagai aturan yang akan menangani skenario penggunaan terhadap serangan yang lebih spesifik, yang mungkin untuk beberapa pengguna namun tidak relevan terhadap pengguna lainnya. Secara *default*, aturan ini sudah diaktifkan ketika menggunakan Falco untuk pertama kali. Gambaran secara umum cara kerja *rule deployment of privileged container* pada Falco sebelum proses *tuning* dapat dilihat diagram YAML pada Gambar 7.

rule	Deployment of Privileged container
condition	container_started and container and container.privileged=true

Gambar 7. Diagram YAML *rule* sebelum *tuning* dengan *Deployment of Privileged Container*

Setelah dilakukan berbagai percobaan serangan dan melihat pola dari tiap-tiap serangan, penelitian ini melakukan modifikasi atau penambahan kondisi pada aturan tersebut untuk meningkatkan kapabilitas aturan Falco sesuai dengan meningkatnya variasi serangan untuk skenario *deployment of privileged container*. Proses *tuning* ini dilakukan untuk mengurangi adanya *false positive* dari aturan ini terhadap pembuatan *container* dengan *privilege* tinggi yang dilakukan oleh tim internal dalam melakukan pengembangan. Maka dari itu, perlu penambahan kondisi pada *macro* "user_privileged_containers", terkait dengan daftar "proc.name" yang diizinkan untuk dijalankan dengan mode *privilege* tinggi milik *developer*. Gambaran secara umum cara kerja *rule deployment of privileged container* pada Falco setelah proses *tuning* dapat dilihat diagram YAML pada Gambar 8.

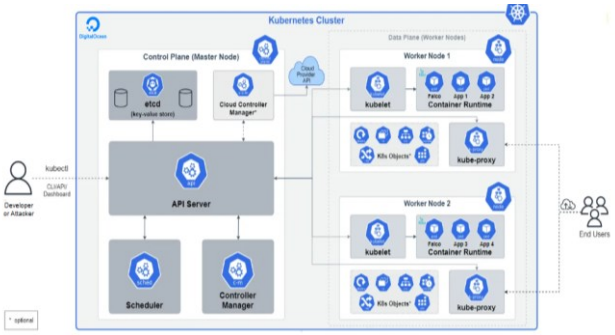
rule	Deployment of Privileged container
condition	container_started and container and container.privileged=true

rule	Exception for Deployment of Developer Privileged container
condition	spawned_process and container and not user_privileged_containers

Gambar 8. Diagram YAML *rule* setelah *tuning* dengan *deployment of privileged container*

C. Pengujian Performa *Rule*

Topologi dari skenario serangan yang akan dilakukan pada penelitian ini dapat dilihat pada Gambar 9. Dalam hal ini, *attacker* sudah mendapat akses terhadap *API Server* dari klaster Kubernetes sehingga dapat mengakses *pod* secara langsung untuk menjalankan *malicious activity* di dalam *pod* tersebut. Di sisi lain, masing-masing *worker node* akan terinstal satu buah agen Falco yang akan melakukan monitoring terhadap *malicious activity* pada *node*-nya.



Gambar 9. Topologi serangan

Pengujian performa *rule* mencakup komparasi dari kapabilitas deteksi Falco menggunakan *rule* sebelum dan setelah *tuning*. Pengujian dilakukan terhadap tiga skenario *malicious activity* yang sebelumnya sudah di-*tuning*, di antaranya *Remote Code Execution, exfiltration using common Linux binaries*, dan *deployment of privileged container.*

1. Skenario *Remote Code Execution*

Skenario *Remote Code Execution* dirancang untuk mengevaluasi efektivitas Falco dalam mendeteksi aktivitas yang terkait dengan eksekusi *Remote Code Execution* (RCE) dalam lingkungan terkontainerisasi. Fokus utamanya adalah mengidentifikasi potensi ancaman keamanan yang timbul dari upaya eksekusi kode yang tidak sah atau berbahaya. Evaluasi ini melibatkan simulasi berbagai teknik eksekusi *Remote Code Execution* (RCE) yang mungkin digunakan oleh penyerang untuk mengompromi aplikasi terkontainerisasi. Skenario ini mempertimbangkan metode umum seperti penggunaan alat seperti Socat, SQLite3, Ncat, Telnet dan bahasa pemrograman seperti Python, Ruby, dan PHP untuk eksekusi kode. Tujuan menguji set aturan Falco secara komprehensif terhadap teknik-teknik *Remote Code Execution* (RCE) adalah untuk memastikan deteksi yang tepat waktu dan akurat, memungkinkan respons proaktif untuk mengurangi risiko keamanan.

2. Skenario *Exfiltration using Common Linux Binaries*

Skenario *exfiltration using common Linux binaries* berfokus untuk mengevaluasi kemampuan deteksi Falco terkait dengan aktivitas eksfiltrasi data menggunakan biner linux yang umum digunakan seperti wget, whois, bash, openssl, curl, KSH, rsync, SCP dan SFTP. Pada dasarnya biner Linux tersebut merupakan biner Linux yang umum digunakan dalam lingkungan Linux, akan tetapi jika disalahgunakan, biner Linux tersebut dapat digunakan untuk melakukan hal-hal yang berbahaya seperti eksfiltrasi data. Eksfiltrasi data merupakan ancaman yang signifikan. Penyerang dapat menggunakan biner Linux umum untuk mengirimkan informasi sensitif keluar dari lingkungan terkontainerisasi.

3. Skenario *Deployment of Privileged Container*

Deployment of privileged container melibatkan pengujian kemampuan deteksi Falco dalam mengidentifikasi pembuatan kontainer dengan *privilege* tinggi. Fokus utamanya adalah pada pemeriksaan dan identifikasi aktivitas berpotensi merugikan dalam kontainer yang memiliki *privilege* yang tinggi. Jika seorang penyerang berhasil membuat dan menggunakan kontainer dengan *privilege* tinggi, hal itu membuka pintu untuk tindakan berbahaya dengan konsekuensi yang besar, seperti akses tidak sah ke data penting, manipulasi pengaturan sistem penting, atau eksploitasi kerentanan.

Pada dasarnya menjalankan *privileged container* dapat mengizinkan tim internal untuk mendapat akses kritikal ke sumber daya yang ada di *host*. Namun, jika disalahgunakan, akses istimewa ini dapat menimbulkan risiko keamanan yang serius. Oleh karena itu, penting untuk memantau pembuatan *privileged container* secara cermat guna meminimalkan kemungkinan terjadinya kejadian yang tidak diinginkan dan mencegah potensi kerentanan keamanan yang dapat dieksploitasi

D. Pengujian *Mean Time To Detect (MTTD)*

Pengujian *Mean Time to Detect* (MTTD) dilakukan untuk menghitung rata-rata waktu yang dibutuhkan sejak sebuah serangan siber terjadi hingga serangan tersebut di deteksi oleh sistem *monitoring* Falco. Untuk melakukan pengujian ini dibutuhkan data berupa waktu ketika serangan dijalankan serta waktu ketika Falco mendeteksi serangan. Kemudian, akan dihitung selisihnya untuk mendapatkan nilai MTTD sejak sebuah serangan siber terjadi hingga serangan tersebut dideteksi oleh sistem monitoring Falco.

Untuk mendapatkan data waktu ketika serangan dijalankan, penelitian ini menggunakan utilitas *echo* dan *date* pada VM penyerang untuk mencatat waktu saat *payload* di jalankan. Penggunaan utilitas *time* dilakukan dengan cara menuliskan *command echo* dan *time* diikuti dengan *payload* serangan yang akan diujikan. Setelah itu, akan tertampil *output* berupa waktu ketika *payload* tersebut dijalankan. Format penggunaan dari utilitas *echo* dan *time* dapat dilihat pada Gambar 10 dengan nilai x diganti dengan *script payload* yang akan digunakan.

```
echo $(date; x;)
```

Gambar 10. Format informasi waktu ketika *payload* dijalankan pada VM penyerang

Sebagai contoh, untuk skenario pengujian *Remote Code Execution* dengan utilitas Socat, dapat dilihat pada Gambar 11.

```
root@privileged-pod:/# echo $(date;socat TCP:157.245.195.228:34223 EXEC:'sh'
,pty,stderr,setsid,sigint,sane)
Mon Jun 10 07:07:13 UTC 2024
```

Gambar 11. Contoh penggunaan format RCE

Untuk mendapatkan data waktu ketika Falco mendeteksi serangan, penelitian ini menggunakan data notifikasi yang berada pada *log* Falco. Jika Falco berhasil mendeteksi adanya *malicious activit*y, maka pada *log* Falco akan tercatat juga waktu ketika Falco mendeteksi serangan tersebut. Dapat dilihat pada Gambar 12 sebagai contoh untuk mengambil informasi waktu pada *log* Falco.

```
07:07:13.216120914: Notice Remote Code Execution (gparent=bash ggparent=cont
ainerd-shim gggparent=systemd fd.sip=<NA> connection=<NA> lport=<NA> rport=<
NA> fd_type=<NA> fd_proto=fd.l4proto evt_type=execve user=root user_uid=0 us
er_loginuid=-1 process=socat proc_exepath=/usr/bin/socat parent=socat comman
d=socat TCP:157.245.195.228:34223 EXEC:sh,pty,stderr,setsid,sigint,sane term
inal=34844 exe_flags=EXE_WRITABLE|EXE_UPPER_LAYER container_id=1c69f5fb61ce
container_image=docker.io/library/nginx container_image_tag=7383c266ef252ad7
0806f3072ee8e63d2a16d1e6bafa6146a2da867fc7c41759 container_name=privileged-p
od k8s_ns=default k8s_pod_name=privileged-pod evt_type=execve)
```

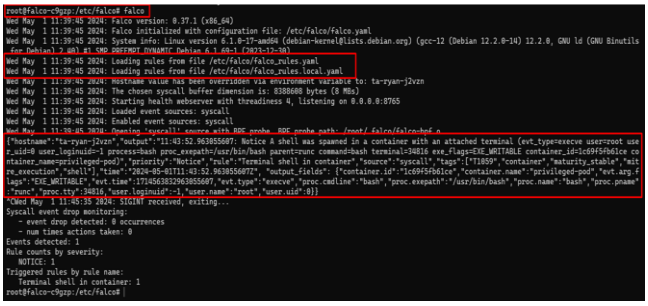
Gambar 12. Informasi waktu pada log Falco

Setelah mendapatkan kedua data tersebut, penelitian ini menghitung selisih dari kedua data tersebut untuk mendapatkan nilai MTTD. Pengujian ini dilakukan untuk melihat komparasi durasi waktu deteksi antara *rule* sebelum dan sesudah dilakukan *tuning*.

IV. HASIL DAN PEMBAHASAN

A. Pengujian Fungsionalitas

Pengujian fungsionalitas sistem dilakukan dengan tujuan untuk memastikan fungsionalitas Falco sudah berjalan dengan baik. Pengujian dilakukan dengan cara memasukkan seluruh *rule* Falco, baik sebelum dilakukan *tuning*, maupun setelah dilakukan *tuning*. Setelah itu, untuk menjalankan Falco dapat dengan menjalankan perintah “falco”. Jika terdapat kesalahan pada penulisan *rule*, maka akan muncul pesan eror yang menunjukkan letak kesalahan penulisan *rule*. Akan tetapi, jika tidak terdapat kesalahan pada penulisan *rule*, maka Falco akan berjalan dan pengguna dapat melihat *output event* yang terdeteksi secara langsung pada *shell*.



Gambar 13. Hasil Pengujian Fungsionalitas Falco

Gambar 13 menunjukkan bahwa ketika menjalankan Falco, tidak terdapat pesan eror. Jika terdapat kesalahan penulisan dalam pembuatan *rules*, Falco akan memberikan pesan eror dan memberitahu pengguna untuk memperbaiki penulisan *rule* tersebut sebelum menjalankan Falco. Selain itu, ketika mencoba melakukan sebuah *malicious activity* dengan cara menjalankan shell pada salah satu *pod*, Falco berhasil mendeteksi *malicious activity* tersebut. Hal ini

menunjukkan bahwa Falco telah berjalan dengan baik dalam mendeteksi *malicious activity* sesuai dengan *rule* yang terdapat didalamnya.

B. Pengujian Performa *Rule*

1. Skenario *Remote Code Execution*

Dari hasil pengujian terhadap aturan Falco sebelum dan sesudah dilakukan proses *tuning*, terlihat pada Tabel 3, setelah dilakukan proses *tuning rule*, terdapat kenaikan persentase kapabilitas deteksi terhadap skenario *Remote Code Execution* dari 44% ke 100%. Dalam artian, setelah dilakukan proses *tuning*, aturan tersebut dapat mendeteksi seluruh skenario pengujian yang dilakukan. Kenaikan persentase ini dicapai dikarenakan proses *tuning* melibatkan berbagai percobaan untuk memahami *signature* dari tiap serangan. Dalam hal ini, skenario *Remote Code Execution* dapat dideteksi berdasarkan "proc.name" dari penggunaan binari Linux yang sering digunakan dalam melakukan *Remote Code Execution*. Binari Linux yang dapat dideteksi dari serangan ini di antaranya *nc*, *ncat*, *socat*, *telnet*, *zsh*, dan *rushcat*. Kondisi "proc.name" ini juga disandingkan dengan operator *boolean* "dan" terhadap makro "spawned_process" yang memiliki kondisi (*evt.type in (execve, execveat) and evt.dir=<*). Makro tersebut memang telah tersedia pada aturan falco dan biasanya penggunaannya disandingkan dengan kondisi "proc.name" untuk mendeteksi eksekusi dari binari Linux yang didefinisikan pada kondisi "proc.name".

Tabel 3. Hasil pengujian performa *rule* dengan RCE

Skenario Pengujian	Hasil Pengujian	
	Sebelum *Tuning*	**Setelah *Tuning***
Socat 1	×	✓
Socat 2	×	✓
Sqlite3 NC MKFIFO	×	✓
NC MKFIFO	×	✓
PHP Shell Exec	✓	✓
Telnet	×	✓
Python	✓	✓
Ruby	✓	✓
ZSH	✓	✓
Total	**4** dari **9** Berhasil	**9** dari **9** Berhasil
Persentase	**44%**	**100%**

2. Skenario *Exfiltration using Common Linux Binaries*

Dari hasil pengujian terhadap aturan Falco sebelum dan sesudah dilakukan proses *tuning*, terlihat pada Tabel 4, setelah dilakukan proses *tuning rule*, terdapat kenaikan persentase kapabilitas deteksi terhadap skenario *exfiltration using common Linux binaries* dari 33% ke 100%. Dalam artian, sebelum dilakukan *tuning* aturan tersebut hanya dapat mendeteksi tiga skenario yang diujikan sedangkan setelah dilakukan proses *tuning*, aturan tersebut dapat mendeteksi seluruh skenario pengujian yang dilakukan. Kenaikan persentase ini dicapai dikarenakan proses *tuning* melibatkan berbagai percobaan untuk memahami *signature* dari tiap serangan. Dalam hal ini, skenario *exfiltration using common Linux binaries* dapat dideteksi berdasarkan proc.name beberapa binary yang termasuk dalam pengujian, di antaranya *wget*, *whois*, *bash*, *openssl*, *curl*, KSH, SCP, *Rsync* dan SFTP. Tiap-tiap binari juga disandingkan dengan parameter *boolean* "dan" terhadap *event* "proc.cmdline" untuk memastikan *event* yang terdeteksi merupakan *event* yang berkaitan dengan kegiatan eksfiltrasi data. Hal ini dikarenakan binari Linux umum tersebut sangat sering dijumpai penggunaannya, tetapi hanya perlu melakukan monitoring terhadap parameter yang terkait dengan proses eksfiltrasi. Sehingga, ketika terdapat penggunaan binari Linux umum yang tidak terkait dengan eksfiltrasi data, maka falco tidak akan mendeteksi nya sebagai *malicious activity*.

Tabel 4. Hasil pengujian performa *rule* dengan *Exfiltration Using Common Linux Binaries*

Skenario Pengujian	Hasil Pengujian	
	Sebelum *Tuning*	**Setelah *Tuning***
wget	×	✓
whois	×	✓
bash	×	✓
openssl	×	✓
curl	×	✓
KSH	×	✓
rsync	✓	✓
SCP	✓	✓
SFTP	✓	✓
Total	**3** dari **9** Berhasil	**3** dari **9** Berhasil
Persentase	**33%**	**100%**

3. Skenario *Deployment of Privileged Container*

Dari hasil pengujian terhadap aturan Falco sebelum dan sesudah dilakukan proses *tuning*, terlihat pada Tabel 5, setelah dilakukan proses *tuning rule*, terdapat kenaikan persentase kapabilitas deteksi terhadap skenario *deployment of privileged container* dari 66% ke 100%. Dalam artian, setelah dilakukan proses *tuning*, aturan tersebut dapat mendeteksi seluruh skenario pengujian yang dilakukan. Kenaikan persentase ini dicapai dikarenakan proses *tuning* melibatkan berbagai percobaan untuk memahami *signature* dari tiap serangan. Dalam hal ini *rule* pada skenario *deployment of privileged container* dapat mengurangi adanya *false positive* dengan mengurangi *alert* yang dihasilkan ketika proses *deployment container* dengan *privilege* tinggi yang dimiliki oleh developer. Hal ini dilakukan dengan cara menambahkan kondisi pengecualian dengan memberikan daftar "proc.name" apa saja yang diizinkan untuk dijalankan dengan *privilege* tinggi.

Tabel 5. Hasil pengujian performa *rule* dengan *Privileged Container*

Skenario Pengujian	Hasil Pengujian	
	Sebelum *Tuning*	**Setelah *Tuning***
Deployment privileged container using command line	✓	✓
Deployment privileged container using manifest file	✓	✓
Deployment developer privileged container (exception)	×	✓
Total	**2** dari **3** Berhasil	**2** dari **3** Berhasil
Persentase	**33%**	**100%**

C. Pengujian *Mean Time To Detect (MTTD)*

Pengujian selanjutnya yaitu pengujian *Mean Time to Detect* (MTTD). Pengujian ini dilakukan untuk menentukan rata-rata waktu yang dibutuhkan dalam mendeteksi sebuah serangan sejak serangan tersebut diluncurkan hingga terdeteksi oleh sistem monitoring Falco.

1. Skenario *Remote Code Execution*

Berdasarkan hasil perhitungan menggunakan rumus MTTD pada Gambar 2, bisa dilihat pada Tabel 6, bahwa tidak terdapat perbedaan signifikan antara MTTD menggunakan *rule* sebelum dan sesudah dilakukan *tuning*. Sebelum dan setelah dilakukan *tuning*, hasil pengujian menunjukkan bahwa rata-rata MTTD untuk semua skenario adalah sama-sama 0 detik. Hal ini menunjukkan respon sistem Falco yang sangat cepat dalam mendeteksi serangan RCE.

Tabel 6. Hasil pengujian MTTD dengan RCE

Skenario Pengujian	Hasil Pengujian (detik)	
	Sebelum *Tuning*	**Setelah *Tuning***
Socat 1	0	0
Socat 2	0	0
Sqlite3 NC MKFIFO	0	0
NC MKFIFO	0	0
PHP Shell Exec	0	0
Telnet	0	0
Python	0	0
Ruby	0	0
ZSH	0	0
MTTD	**0**	**0**

2. Skenario *Exfiltration using Common Linux Binaries*

Berdasarkan hasil perhitungan menggunakan rumus MTTD pada Gambar 2, bisa dilihat pada Tabel 7, hasil perhitungan *Mean Time to Detect* (MTTD) pada skenario *exfiltration using common Linux binaries* sebelum dan setelah *tuning*, didapatkan nilai MTTD dengan angka yang sama sebesar 0 detik. Hasil ini menunjukkan respons yang cepat dari sistem monitoring Falco terhadap serangan yang dilakukan menggunakan berbagai binari Linux seperti *wget*, *whois*, *bash*, *openssl*, *curl*, KSH, *Rsync*, SCP dan SFTP.

Tabel 7. Hasil pengujian MTTD dengan *Exfiltration Using Common Linux Binaries*

Skenario Pengujian	Hasil Pengujian (detik)	
	Sebelum *Tuning*	**Setelah *Tuning***
wget	0	0
whois	0	0
bash	0	0
openssl	0	0
curl	0	0
KSH	0	0
rsync	0	0
SCP	0	0
SFTP	0	0
MTTD	**0**	**0**

3. Skenario *Deployment of Privileged Container*

Berdasarkan hasil perhitungan menggunakan rumus MTTD pada Persamaan (1), hasil perhitungan MTTD pada Tabel 8 menunjukkan bahwa tidak terdapat perbedaan signifikan antara MTTD menggunakan *rule* sebelum dan sesudah dilakukan *tuning*. Sebelum dilakukan *tuning*, hasil pengujian menunjukkan bahwa rata-rata MTTD untuk semua skenario *privileged container* adalah 2,67 detik. Setelah dilakukan *tuning*, rata-rata MTTD tercatat di angka 4,33 detik untuk semua skenario. Perbandingan antara rata-rata MTTD sebelum dan setelah *tuning* menunjukkan bahwa tidak ada perbedaan yang signifikan. Hal ini menunjukkan efektivitas sistem Falco yang baik dalam mendeteksi *malicious activity*.

Jika dibandingkan dengan skenario *Remote Code Execution* (RCE) dan *exfiltration using common Linux binaries*, proses pendeteksian pada skenario *deployment of privileged container* memerlukan waktu lebih lama sekitar 2-4 detik. Hal ini disebabkan karena pada proses *deployment privileged pod*, dibutuhkan waktu tambahan yang dibutuhkan oleh Kubernetes untuk dapat menjalankan *container* tersebut dari proses *ContainerCreating* hingga proses *running*. Meskipun demikian, hasil perhitungan menunjukkan bahwa waktu yang dibutuhkan Falco untuk mendeteksi adanya *malicious activity* di lingkungan Kubernetes masih relatif singkat. Hal ini menunjukkan efektivitas yang baik dari sistem Falco dalam mendeteksi aktivitas berbahaya yang memungkinkan proses *incident handling* bisa lebih cepat untuk meminimalisir resiko yang dapat timbul.

Tabel 8. Hasil pengujian MTTD dengan *Privileged Container*

Skenario Pengujian	Hasil Pengujian (detik)	
	Sebelum *Tuning*	**Setelah *Tuning***
Deployment privileged container using command line	4	0
Deployment privileged container using manifest file	4	0
Deployment developer privileged container (exception)	0	0
Total	**2,67**	**0**

V. SIMPULAN

Berdasarkan data yang diperoleh dan analisis yang dilakukan dari penelitian ini, dapat diambil beberapa kesimpulan. Pertama, implementasi Falco dapat menjawab masalah diperlukan adanya monitoring pada fase *runtime* untuk mendeteksi *malicious activity* di Kubernetes. Selanjutnya, berdasarkan hasil pengujian performa *rule*, proses *tuning rule* yang telah dilakukan berhasil meningkatkan kapabilitas deteksi skenario *Remote Code Execution* sebesar 56%, skenario *exfiltration using common Linux binaries* sebesar 67%, dan skenario *deployment privileged container* sebesar 33%. Terakhir, berdasarkan data yang diperoleh dari pengujian skenario *Remote Code Execution* (RCE), *exfiltration using common Linux binaries*, dan *deployment privileged container*, tidak terdapat perbedaan durasi deteksi serangan sebelum dan setelah proses *tuning*. Rata-rata nilai MTTD atau waktu yang dibutuhkan untuk mendeteksi serangan tetap konsisten di bawah lima detik, menunjukkan respons cepat dari sistem *monitoring* Falco dalam menghadapi berbagai serangan pada lingkungan Kubernetes. Hal ini menunjukkan bahwa implementasi Falco dan proses *tuning rule* secara efektif meningkatkan kapabilitas deteksi tanpa mengorbankan waktu deteksi yang cepat.

REFERENSI

[1] B. Yuen, A. Matyushentsev, T. Ekenstam, and J. Suen, GitOps and Kubernetes Continuous Deployment with Argo CD, Jenkins X, and Flux. 2021.

[2] M. McLennan, SK Group, and Zurich Insurance Group, “The Global Risks Report 2022 17th Edition,” 2022.

[3] Redhat, “State of Kubernetes security report,” 2022. Accessed: May 03, 2024. [Online]. Available: https://www.redhat.com/rhdc/managed-files/cl-state-of-kubernetes-security-report-2022-ebook-f31209-202205-en.pdf

[4] S. Singh, “Cloud Computing : An Overview,” Int J Sci Res Sci Eng Technol, pp. 63–70, Jan. 2019, doi: 10.32628/IJSRSET196120.

[5] H. Gantikow, C. Reich, M. Knahl, and N. Clarke, “Rule-Based Security Monitoring of Containerized Environments,” 2020, pp. 66–86. doi: 10.1007/978-3-030-49432-2_4.

[6] Nutanix, “What is Kubernetes?” Accessed: May 13, 2024. [Online]. Available: https://www.nutanix.com/info/what-is-kubernetes#definition

[7] Abhilash, “Self-managed Kubernetes Vs. Kubernetes-as-a-service (Managed Kubernetes).” Accessed: Apr. 25, 2024. [Online]. Available: https://www.ozone.one/self-managed-kubernetes-vs-kubernetes-as-a-service-managed-kubernetes/

[8] A. Y. Wong, E. G. Chekole, and M. O. and J. Zhou, “Threat Modeling and Security Analysis of Containers: A Survey,” arXiv preprint arXiv:2111.11475, Nov. 2021. Accessed: May 25, 2025. [Online]. Available: https://arxiv.org/abs/2111.11475

[9] M. Sroor, R. Mohanani, R. Colomo-Palacios, S. Dasanayake, and T. Mikkonen, “Managing Security Issues in Software Containers: From Practitioners Perspective,” arXiv preprint arXiv:2504.07707, Apr. 2025. Accessed: May 25, 2025. [Online]. Available: https://arxiv.org/abs/2504.07707

[10] GeeksForGeeks, “Introduction of System Call.” Accessed: May 13, 2024. [Online]. Available: https://www.geeksforgeeks.org/introduction-of-system-call/

[11] A. Bansal, R. Awasthi, and A. Venkatrao, “Tracing System Calls Using eBPF - Part 1 | Falco.” Accessed: May 03, 2024. [Online]. Available: https://falco.org/blog/tracing-syscalls-using-ebpf-part-1/

[12] S. Das, “(14) What is Cloud Native & CNCF? Graduated Projects? Info Insight! | LinkedIn.” Accessed: May 03, 2024. [Online]. Available: https://www.linkedin.com/pulse/what-cloud-native-cncf-graduated-projects-info-insight-sandip-das-nhy9e/

[13] O. Yaşar, “What is Falco?” Accessed: May 13, 2024. [Online]. Available: https://medium.com/adessoturkey/what-is-falco-837298a066ee

[14] Plutora, “MTTD (Mean Time to Detect): Defined and Explained.” Accessed: May 03, 2024. [Online]. Available: https://www.plutora.com/blog/mttd-mean-time-to-detect-defined-explained

[15] Falco Authors, “Supported Fields for Conditions and Outputs,” The Falco Project, 2023. Accessed: May 25, 2025. [Online]. Available: https://falco.org/docs/reference/rules/supported-fields/