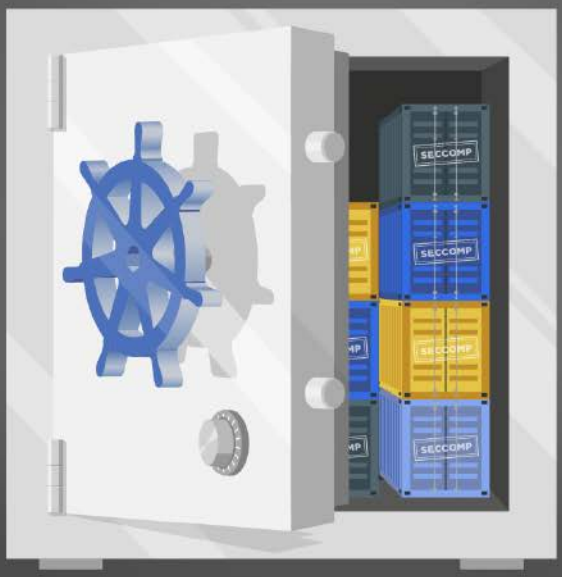
Kubernetes mit Capabilities, SECCOMP und SELinux

Capabilities, SELinux und SECCOMP im Linux-Kernel sollen verhindern, dass Programme verbotene Dinge tun. Kubernetes kann die Methoden nutzen, um den Kernel und andere Prozesse auf dem Server (Node) vor Angriffen zu schützen.

Von Christoph Puppe und Jan Walther



Die Sicherheitskomponenten Capabilities, SELinux und SECCOMP im Linux-Kernel gehören zu den Schutzmaßnahmen, die jeder jederzeit einsetzen kann, die aber aufgrund der Komplexität der Konfiguration nur selten wirklich zum Einsatz kommen. Durch entsprechende Regelwerke sollen sie verhindern, dass Programme Dinge tun, die sie nicht dürfen. Auch Kubernetes kann die Techniken einsetzen, um den Kernel vor Angriffen aus den Containern heraus zu schützen und diesen Containern je nach Szenario bestimmte sicherheitskritische Aktionen zu verbieten, etwa das Erzeugen von TCP-Paketen: Normale TCP-Verbindungen sollen den Containern gestattet bleiben, nicht aber mit Paketen, die sie selbst erstellt haben.

Der Artikel wird zuerst Capabilities, SECCOMP und die zugrunde liegenden Prinzipien erklären und dann die Integration in den Betrieb von Containern mit Kubernetes aufzeigen. Ein Verständnis von SELinux wird hier vorausgesetzt, iX hat in den vergangenen Jahren viele Artikel dazu veröffentlicht und bietet auch Workshops an (siehe ix.de/zhxj).

Ein zentraler Fähigkeitsnachweis: Capabilities

Für Berechtigungen in Linux gibt es zwei grundlegende Konzepte: Neben den seit Urzeiten vorhandenen dateisystembasierten Berechtigungen (rwx) enthält der Kernel seit Version 2.2 die Capabilities (deutsch: Fähigkeiten). Sie erlauben es, die Berechtigungen der Benutzer – inklusive Root – detailliert einzuschränken. Eine Übersicht aller Capabilities und in welcher Kernel-Version die Entwickler sie jeweils eingeführt haben, liefert der Aufruf

```
man 7 capabilities
```

Eine Capability ist die Berechtigung, Aufrufe des Kernels (Syscalls) durchzuführen. Alle Capabilities haben die Form `CAP_` gefolgt von einem ein- oder mehrteiligen Namen jeweils in Großbuchstaben und mit Unterstrichen getrennt. Ein Beispiel für die Verwendung von Capabilities auf modernen Systemen ist das Netzwerktool ping. Auf manchen Systemen, die den Kernelparameter `net.ipv4.ping_group_range` nicht geändert haben, müssen User (root oder nicht-root) für einen Aufruf die Capability haben, beliebig Pakete abzuschicken (`CAP_NET_RAW` beziehungsweise `CAP_NET_ADMIN`). Der Root-Benutzer erhält diese normalerweise automatisch.

Vor der Einführung von Capabilities setzten viele Distributionen das `setuid`-Bit beim ping-Binary, damit auch normale Benutzer es aufrufen konnten. Führt ein Benutzer eine Datei mit dem `setuid`-Bit aus, wird statt der ID des Benutzers die ID des Besitzers der Datei (üblicherweise root) als effektive Benutzer-ID gesetzt, mit dem Risiko einer Privilege Escalation bei Fehlern im Programm.

Der Kasten „Capabilities im Linux-Kernel“ listet die Capabilities mit einer verkürzten Beschreibung. Es spricht in den meisten Fällen nichts dagegen, die komplette Liste für Prozesse ohne administrative Aufgaben zu verbieten, etwa für Server in einem Container. Sie enthält allerdings auch `CHOWN`, `KILL`, `MKNOD` und `SETUID`, die Docker und containerd standardmäßig erlauben. Meist gibt es aber keinen Grund, warum ein Prozess einen anderen Prozess abschießen, eine Gerätedatei anlegen, die UID setzen oder den Inhaber einer Datei ändern können sollte.

iX-TRACT

- SELinux und SECCOMP bieten guten Schutz, sofern sie ihre Fähigkeiten auch ausspielen dürfen.
- Die Konfiguration der jeweiligen Profile ist aufwendig und nicht ohne Fallstricke.
- Für den Betrieb mit allen Fähigkeiten ist eine Automatisierung in der CI/CD unumgänglich.

Problemfall User-Namespace

Das Bereitstellen von Kernel-Ressourcen, die normalerweise Admin-Rechte benötigen, in Namespaces hat die Kernel-Entwickler vor ein großes Problem gestellt. Neben dem offensichtlichen Aufwand, ein

Capabilities im Linux-Kernel

- AUDIT_CONTROL – Kernel audit log kontrollieren
- AUDIT_READ – Kernel audit log lesen
- AUDIT_WRITE – Kernel audit log schreiben
- BLOCK_SUSPEND – das System vom Schlafen abhalten
- BPF – mit den privilegierten BPF-Systemen des Kernels interagieren
- CHECKPOINT_RESTORE – Zugriff aufs Checkpoint-Subsystem des Kernels
- CHOWN – beliebige Benutzerwechsel durchführen
- DAC_OVERRIDE – alle Dateiberechtigungen ignorieren
- DAC_READ_SEARCH – Dateiberechtigungen für Lesen und Ausführen ignorieren
- FOWNER – Metadaten für Dateien bearbeiten
- FSETID – UID und GID nach Bearbeiten beibehalten
- IPC_LOCK – Speicher blocken oder per Huge Pages reservieren
- IPC_OWNER – Berechtigungen für System V IPC ignorieren
- KILL – beliebige Signale senden (etwa `SIGKILL`)
- LEASE – Leases auf beliebige Dateien erstellen
- LINUX_IMMUTABLE – Dateien als unveränderlich markieren
- MAC_ADMIN – SMACK administrieren
- MAC_OVERRIDE – SMACK-Zugriffsrechte überschreiben
- MKNOD – spezielle Dateien erstellen
- NET_ADMIN – Netzwerkadministration
- NET_BIND_SERVICE – privilegierte Ports verwenden
- NET_BROADCAST – nicht verwendet
- NET_RAW – RAW- und PACKET-Sockets erstellen
- PERFMON – Performancemonitoring
- SETGID – GID setzen
- SETFCAP – Datei-Capabilities setzen
- SETPCAP – Prozess-Capabilities setzen
- SETUID – UID setzen
- SYS_ADMIN – weitgehende Administratorrechte
- SYS_BOOT – das System neu starten oder kexec ausführen
- SYS_CHROOT – chroot(2) verwenden und mount-Namespaces erstellen
- SYS_MODULE – Kernel-Module verwalten
- SYS_NICE – Scheduler verwalten
- SYS_PACCT – Accounting für Prozesse verwalten
- SYS_PTRACE – ptrace(2) auf beliebige Prozesse
- SYS_RAWIO – Low-Level I/O
- SYS_RESOURCE – Zugriff auf Low-Level-Ressourcen
- SYS_TIME – Systemzeit verwalten
- SYS_TTY_CONFIG – virtuelle Terminals verwalten
- SYSLOG – Syslog verwalten
- WAKE_ALARM – Alarme verwalten, die das System aufwecken

völlig neues Berechtigungskonzept einzuführen, können Prozesse, die nicht als Root laufen, jetzt Codeabschnitte des Kernels erreichen, die vorher ausschließlich Root vorbehalten waren. Wo eine Schwachstelle in diesen Abschnitten vorher ein Ärgernis war, kann es nun zu einer Eskalation von Privilegien kommen.

Ein Administrator, der sich durch eine Schwachstelle Administratorrechte sichern kann, ist kaum eine Schlagzeile wert. Ein normaler Benutzer, der über Netzwerk- und Benutzer-Namespace jedoch Root-Rechte auf der Maschine erlangt, ist hingegen ein Problem. Der für Google arbeitende Sicherheitsforscher Andy Nguyen (theflow) zeigte diesen Vorgang eindrucksvoll in CVE-2021-22555. Bei über 300 Syscalls, die der Linux-Kernel bereitstellt, ist die Angriffsoberfläche entsprechend groß.

SECCOMP filtert alle Syscalls

Der Secure Computing Mode SECCOMP ist eine schon sehr lange bekannte, effektive und relativ einfach einzusetzende Sicherheitsmaßnahme, bei der ein Prozess nur noch definierte Syscalls anwenden kann. Die darf – wie SELinux auch – leider nur sehr selten wirklich zeigen, was sie kann. Bei beiden ist es üblich, Profile zu erstellen, die zwar das Schlimmste verhindern, aber so weit gefasst sind, dass sie keinen Prozess behindern. Das von Docker mitgelieferte Profil verbietet 44 von über 300 Syscalls. Das ist besser als nichts, aber weit weg von dem, was möglich wäre, da beispielsweise ein Apache-Webserver nur sechs Syscalls benötigt. Das Standardprofil lässt in diesem Beispiel rund 250 Syscalls offen, die auch geschlossen sein könnten und die ein Angreifer im Kontext des Apache-Prozesses aufrufen kann. Das mag bei der Compliance gut aussehen, weil der Auditor bei der Frage „SELinux und SECCOMP werden eingesetzt“ in seiner Tabelle ein Ja ankreuzen darf, aber wirklich viel Sicherheit ist mit diesen minimalen Profilen nicht gewonnen.

Wenn der Kernel einen neuen Prozess ausführt, ist dieser in seinen Syscalls normalerweise nicht eingeschränkt. Erst ein

Listing 1: Beispiel für das Setzen von Capabilitites

```
apiVersion: v1
kind: Pod
metadata:
 name: security-context-demo-4
spec:
 containers:
 - name: sec-ctx-4
   image: gcr.io/google-samples/node-hello:1.0
   securityContext:
     capabilities:
       add: ["NET_ADMIN", "SYS_TIME"]
```

expliziter Aufruf des Syscall `seccomp` setzt den Status des Prozesses entweder in

- `SECCOMP_SET_MODE_STRICT` – der Prozess darf nur noch bereits geöffnete Dateien lesen und schreiben und sich selbst beenden – oder
- `SECCOMP_SET_MODE_FILTER` – hier wird als Argument eine Definition des Filters der erlaubten Syscalls und ihrer Argumente übergeben. Diese wird im Kontext von Kubernetes auch als SECCOMP-Profil bezeichnet.

Weitere Details dazu liefert die Manpage zu `seccomp(2)` (siehe ix.de/zhxj). Die Idee ist es, jedem Prozess nur die Aktionen – also Syscalls – zu erlauben, die er auch für seine Arbeit braucht. So lassen sich viele Aktionen verhindern, die Angreifern weitere Zugriffe auf andere Prozesse, das Dateisystem oder das Netzwerk verschaffen könnten. Die dem Prozess so auferlegten Einschränkungen werden dann auch über `clone(2)`, `fork(2)` und `execve(2)` vererbt – jedenfalls, solange die Syscalls im ursprünglichen SECCOMP-Profil erlaubt waren.

SECCOMP und insbesondere die für den BPF (Berkeley Packet Filter) geschriebenen Filter erhalten auch Zugriff auf die Parameter des Syscalls, können jedoch keine Zeiger dereferenzieren. Daher lässt sich `open(2)` nicht anhand eines Dateinamens blockieren, da Zeichenketten grundsätzlich als Zeiger übergeben werden. Dies ist eine bewusst getroffene Entscheidung der Kernel-Entwickler, um sogenannte Time-of-Check-to-Time-of-Use-Attacken (TOCTOU) zu verhindern. Dabei werden Parameter zwischen Prüfung und Verwendung verändert, um Sicherheitsmaßnahmen zu entgehen.

SECCOMP, SELinux und die Container

Ein Container ist erst mal nichts weiter als ein weiterer Prozess im User Space des Kernels. Das bedeutet, dass Syscalls aus dem Container den Kernel des Hosts erreichen und dieser sie ausführt, also auch die eventuell darin enthaltenen Angriffe. Die Container-Runtime (CRI, etwa runc, dockerd, containerd, Podman, CRI-O und andere) betreibt diesen Prozess in einer eigenen cgroup, damit in einem eigenen Namespace und mit einer Benutzer-ID, die nie 0 sein sollte, es aber manchmal sein muss – bei Containern für Aufgaben der Systemverwaltung. Der von der CRI gestartete Prozess kann vor dem Ausführen des eigentlichen Images

- Capabilities entzogen bekommen oder zusätzliche erhalten,
- in den Modud `SECCOMP Filtered` versetzt oder
- mit SELinux-Profil gestartet werden.

Kommen dabei detaillierte Profile und Filter zum Einsatz, ist der Prozess gut gegen Angriffe geschützt. Wer Kata oder Firecracker einsetzt, geht hier einen anderen Weg, da diese Runtimes einen eigenen Kernel im Container starten, der dann das Image ausführt. Eine derart virtualisierte Umgebung schottet das Image besser ab, als dies cgroups und SECCOMP ermöglichen [1].

Damit die CRI ein SECCOMP-Profil und eine SELinux Policy anwendet, müssen diese beim Aufruf als Parameter mit übergeben werden. Nur selten geschieht das Starten eines Containers per Hand oder systemd. Die übliche Variante ist, mehrere Server mit Kubernetes in einem Cluster zusammenzufassen und dem Kubelet die Kontrolle über die Container auf dem Server zu geben. Capabilities werden in Kubernetes direkt über das Feld `securityContext` im Manifest des Pods (Zusammenfassung mehrerer Container in einem Namespace) hinzugefügt (`add`) oder entfernt (`drop`). Dabei erwartet dieses Feld den Namen aller Capabilities ohne das vorangestellte `CAP_`. Aus `CAP_KILL` wird so zum Beispiel `KILL`.

Listing 1 zeigt ein Beispiel aus der Kubernetes-Dokumentation für das Setzen von Capabilities. Zu beachten ist hier, dass ein clusterweites oder pro Node definiertes Profil schon mit dem Feature `SeccompDefault` des Kubelet gesetzt sein kann. Die Kurzform, um alle Capabilities zu entfernen, ist

```
CONTAINERS[] .SECURITYCONTEXT .↵
    CAPABILITIES .DROP | INDEX("ALL")
```

Wenn der Prozess dann doch welche benötigt, kann das Manifest diese definieren. Listing 1 fügt diesem Container zwei Capabilities hinzu, die eigentlich für jeden Container verboten sein sollten, da sie ihm stark erhöhte Rechte einräumen.

Listing 2 stammt ebenfalls aus der K8s-Dokumentation und zeigt die Definition der erlaubten Syscalls. Alle anderen werden mit der `defaultAction` behandelt und sind in diesem Beispiel verboten. Konkret erhalten die Prozesse beim Versuch, einen verbotenen Syscall aufzurufen, einen Fehlercode, können aber im Zweifelsfall weiterarbeiten. Alternativ kann aber auch `SCMP_ACT_LOG` gesetzt werden, um Probleme nur zu protokollieren. `SCMP_ACT_KILL_PROCESS` beendet hingegen den Prozess. Die Datei dient zur Verwendung als Profil in einem Kubelet, muss also als Datei auf dem Node liegen. Wie sie dorthin kommt, ist eine Frage, die das Betriebsteam klären muss. Dieses Profil steht dann in der Definition eines Pods und aktiviert die Datei. Listing 3 zeigt den Aufruf aus einem Manifest.

Noch mehr Sicherheit mit SELinux

Auch die SELinux Policy wird einmal als Default für den ganzen Server gesetzt und dann im `securityContext` für jeden

Listing 2: SECCOMP Policy ERRNO.json

```
{
    "defaultAction": "SCMP_ACT_ERRNO",
    "architectures": [
        "SCMP_ARCH_X86_64",
        "SCMP_ARCH_X86",
        "SCMP_ARCH_X32"
    ],
    "syscalls": [
        {
            "names": [
                "accept4",
                "epoll_wait",
                "futex",
                "madvise",
                "epoll_ctl",
                "vfork",
                "mmap",
                "read",
                "write",
                "close",
                "sched_getaffinity",
                "munmap",
                "brk",
                "rt_sigaction",
                "clone",
                "bind",
                "socket",
                "writev"
            ],
            "action": "SCMP_ACT_ALLOW"
        }
    ]
}
```

Listing 3: SECCOMP im Pod-Manifest

```
...
 securityContext:
   seccompProfile:
     type: Localhost
     localhostProfile: profiles/ERRNO.json
 containers:
...
```

Listing 4: Ergänzende SELinux-Optionen pro Container

```
...
securityContext:
 seLinuxOptions:
  user: system_u
  role: system_r
  type: container_t
  level: s0:c123,c456
```

Pod oder Container angepasst. Wobei die Einstellungen für den Container diejenigen im Pod überschreiben. Die Kubernetes-API erlaubt unter `SELinuxOptions` dabei das Setzen von `level`, `role` (das SELinux-Label des Prozesses), `type` und `user` (siehe Listing 4).

Bei SELinux kann man aus den Protokollen eines nur im Audit-Modus laufenden Systems mit `sepolicy generate` oder mit Red Hats Tool für Container `udica` eine Policy erstellen. Listing 5 zeigt ein Beispiel für Letzteres.

Das Beispiel beschreibt das manuelle Starten des Containers. Nun soll aber K8s diese übernehmen, das heißt, das Manifest muss den Typ `seLinuxOptions` enthalten (siehe Listing 6).

Die von `udica` erstellten Policies sind mit `semodule` auf den Worker-Nodes zuerst zu installieren. Diese Vorbereitung übernimmt K8s nicht. Die Automatisierung etwa mit einem Job auf den Nodes, der den Node entsprechend als Taint labelt, damit die Pods nur auf Nodes landen, die auch die notwendigen SELinux-Policies enthalten, muss das Betriebsteam selbst entwickeln, testen und ausrollen.

Sperren von Syscalls mit Sicherheitsprodukten

Bei diesen Vorgehensweisen wird deutlich, dass das Thema für K8s noch recht frisch ist. Hier sind kommerzielle Sicherheitsprodukte im Vorteil, da sie über viele Jahre Erfahrung verfügen und keine manuellen Eingriffe oder selbst zu erstellenden Automatisierungen benötigen.

Die Alternative zu SECCOMP ist das Sperren von Syscalls für die Container mit Produkten wie AquaSec oder Sysdigs Falco, die Prozesse während der Ausführung so detailliert beobachten, dass sie erkennen, welche Syscalls der Prozess tatsächlich ausführt. Anhand dieser Informationen erstellen sie dann Profile, die sie mit eigenen Agenten auf den Nodes überwachen und durchsetzen.

Zusätzliche Optionen des Security Context

Die Struktur des `securityContext` stellt dabei noch weitere Einstellmöglichkeiten bereit, so zum Beispiel auch das Flag `allowPrivilegeEscalation`. Das erlaubt es, das `no_new_privs`-Bit für den Container zu setzen. Bei gesetztem Bit garantiert der Kernel, dass `execve` keine neuen Rechte hinzufügt. Berechtigungen etwa aus dem `setuid`-Bit oder dateigebundenen Capabilities werden so ignoriert.

Diese Garantie gilt allerdings nur für `execve`. Besitzt der Prozess jedoch `CAP_SETUID`, kann er weiterhin seine UID nach Belieben setzen. Es gibt bereits einige Bestandteile des Kernels, die unprivilegierte Benutzer nur mit einem gesetzten `no_new_privs`-Bit erreichen können und in Zukunft könnten die Ent-

Listing 5: Policy erstellen mit udica und podman

```
$ podman inspect $CONTAINER_ID > container.json
$ udica -j container.json  some_container
Policy some_container with container id 37a3635afb8f created!

Please load these modules using:
# semodule -i some_container.cil /usr/share/udica/templates/{base_container.cil,net_container.cil,home_container.cil}

Restart the container with: "--security-opt label=type:some_container.process" parameter
```

wickler diesen Schutz auf weitere Teile ausweiten. Administratoren sollten daher schon möglichst jetzt das Bit für ihre Applikationen setzen, um die Angriffsoberfläche innerhalb der Applikationen deutlich zu reduzieren. Eine Liste aller Optionen findet sich in der K8s-Onlinedokumentation (siehe ix.de/zhxj).

Automatisierung der Sicherheit und CI/CD

Warum nutzt kaum jemand SECCOMP und SELinux so, wie es gedacht war, und baut Profile, die Prozesse auf das absolute Minimum reduzieren? Die Antwort ist: Zeit, Komplexität, Druck der Projektleitung, schnell fertig zu werden, und das Risiko für den Betrieb. Wenn nur eine Capability oder Berechtigung in SELinux fehlt, wird der Prozess eine kryptische Fehlermeldung produzieren oder sogar abstürzen. Und jede Codeänderung kann erfordern, dass jemand das Profil überarbeitet, damit die Anwendung weiterhin fehlerfrei läuft. Die Verantwortung, zu spezifizieren, welche Capabilities eine Anwendung benötigt, sollte dabei der Hersteller oder Entwickler tragen. Einige Projekte, wie Istio, gehen hier mit gutem Beispiel voran (siehe ix.de/zhxj).

Für das automatisierte Erstellen von SELinux-Profilen bietet sich `udica` an. Für Capabilities gibt es `capable` aus dem BCC-Projekt (BPF Compiler Collection, siehe ix.de/zhxj). Es kann Capability-Prüfungen im Kernel nachvollziehen. Die Auswertung ist jedoch nicht ganz trivial, da einzelne Capabilities nicht unbedingt direkt aus dem Code der Anwendung angefragt werden. In anderen Fällen ist eine fehlgeschlagene Prüfung auch verkraftbar oder harmlos. So wird etwa die gefährliche Capability `CAP_SYS_ADMIN` geprüft, wenn es darum geht, ob Speicher übercommittet werden darf.

Lising 6: Manifest mit seLinuxOptions

```
apiVersion: v1
kind: Pod
...
spec:
  containers:
  - name: some_container
    image: ...
    securityContext:
      seLinuxOptions:
        type: some_container.process
...
```

Für SECCOMP sind die vom Prozess benutzten Syscalls zu extrahieren. Hier kann das Linux-Bordmittel `strace` helfen, mit dem sich die Systemaufrufe eines neuen oder laufenden Prozesses beobachten lassen. Das zu testende Programm sollte dabei jedoch eine Testsuite durchlaufen, da die kritischen Syscalls sonst eventuell gar nicht protokolliert werden. Ein anderer Weg ist die Analyse des Sourcecodes mit Sourcalyzer für C, was wohl wegen des erforderlichen Aufwands kaum jemand für die Anwendungen in seinen Containern nutzt. Die Beobachtung des Containers ist daher der übliche Weg. Es ist dabei darauf zu achten, dass der Container während der Beobachtung alle Syscalls mindestens einmal aufruft. Automatisierte Funktionstests verhindern es, aus Versehen eine Funktion des Containers während der Beobachtung zu vergessen, sodass dann einer in der Liste der erlaubten Syscalls fehlt.

CI/CD kann das Erstellen von Policies automatisieren. Generiert in der Pipeline für ein Deployment eines Images ein Schritt die Policy aus dem Image, ist es einfacher, diese Policy als Teil der Version im Git abzuspeichern und auf die gewünschten Nodes auszubringen. Auch hier gilt wieder die Einschränkung, dass K8s einem bei der Hinterlegung der Policies nicht die Arbeit abnimmt. Nur bei Capabilities sind die Optionen im Manifest selbst enthalten und benötigen keine Vorbereitung des Nodes. Wer also mehr will als die Capabilities, hat hier einen erheblichen Aufwand. Nur die kommerziellen Werkzeuge erlauben es, Syscalls zu filtern, ohne selbst am Node zu schrauben oder dies zu automatisieren.

Fazit

Capabilities, SECCOMP und SELinux sind ungeliebte, nur selten genutzte Sicherheitsfeatures. Ein automatisiertes Erstellen der Profile und Policies ermöglicht sehr detaillierte Einstellungen. Darüber lässt sich zusammen mit cgroups eine Absicherung der Container realisieren, die weit über das sonst Übliche hinausgeht. Die alte Frage, ob Virtualisierung oder Container sicherer ist, bekommt mit den Möglichkeiten von SECCOMP, SELinux und cgroups eine neue Wendung, denn kaum jemand sichert die Anwendungen auf den virtualisierten Servern so gut ab, wie dieses Triumvirat es kann.

Allerdings bringen sie ohne präzise Profile und Policies keinen Nutzen, und Fehler können schnell auch die Verfügbarkeit negativ beeinflussen. Ohne Automatisierung ist ihr Einsatz daher fast nicht möglich. Ob man die Automatisierung mit Open-Source- oder kommerziellen Werkzeugen realisiert, ist Geschmackssache, wobei die kommerziellen Werkzeuge hier den Open-Source-Tools oft überlegen sind. (avr@ix.de)

Quellen

[1] Udo Seidel; Anwendungen abschotten mit Kata Containers, gVisor und Nabla; iX 11/2018, S. 106

[2] Weitere Informationen zu Hintergründen, Tools und auch zum SELinux-Workshop: ix.de/zhxj

CHRISTOPH PUPPE

ist Managing IT Security Architect und Auditteamleiter für ISO 27001 nach Grundschutz bei SVA System Vertrieb Alexander GmbH, Mitautor des Grundschutz-Kompendiums und ehemaliger Penetrationstester.

JAN WALTHER

beschäftigt sich sowohl akademisch als auch auf der Arbeit als System Engineer bei der SVA GmbH mit Themen rund um Linux, Security und Reverse Engineering.