Mit Go sicher in die Cloud

Die Programmiersprache Go bietet grundlegende Konzepte, die Angriffe auf die Supply Chain erschweren.

Von Bernhard Saumweber

Zeitgemäße Softwareentwicklung ist auf Zusammenarbeit ausgelegt: Anwendungen enthalten Abhängigkeiten auf externe, meist quelloffene Softwarekomponenten und werden in Containern angeliefert, die weitere Software enthalten. Damit steigt die Gefahr für Angriffe in der Lieferkette. Diverse Konzepte von Go mindern die Gefahren und machen die Programmiersprache und ihr Ökosystem zu einer guten Wahl für Cloud-Anwendungen.

Angriffe auf die Lieferkette

Vom Code bis zum aktiven Microservice in der Cloud durchläuft eine Software zahlreiche Schritte und ist abhängig von Drittsystemen und Zulieferungen (siehe Abbildung 1). Der Code der eigentlichen Anwendung liegt typischerweise ebenso in einem vertrauenswürdigen Versionskontrollsystem wie Referenzen auf die benötigten direkten Abhängigkeiten. Der Code der direkten und transitiven Dependencies wird jedoch erst zur Build-Zeit abgerufen. Das Gleiche gilt für viele Build-Tools und im Fall von Cloud-Native-Anwendungen für die Container-Basis-Images und die darin enthaltene Software. Jede Abhängigkeit birgt ein gewisses Risiko und stellt letztlich ein Vertrauensverhältnis dar.

Der vermutlich häufigste Störfall ist eine Schwachstelle in einer Abhängigkeit, die in die eigene Software übernommen wird. Oftmals ist es eine transitive Abhängigkeit, die nicht unmittelbar aus dem Code hervorgeht. Für Teams, die nicht aktiv auf solche Schwachstellen prüfen und die Dependencies regelmäßig aktualisieren, ist es nur eine Frage der Zeit, bis die Lücke ausgenutzt wird. Ein populäres Beispiel ist log4j (der Link zu den Berichten über die Schwachstellen und weitere Ressourcen finden sich unter ix.de/zzdn).

Viele Build-Systeme erlauben daher automatische Updates von Dependencies, sobald eine neue Minor- oder Patch-Version verfügbar ist. Diese an sich gute Idee ermöglicht aber eine neue Art von Angriffen auf die Lieferkette: Es existieren zahlreiche Pakete, die in einer neueren Version absichtlich oder unabsichtlich Schadsoftware enthalten. Viele Entwicklerinnen und Entwickler dürften daher die Befehle zum Durchsuchen ihres Abhängigkeitsbaums inzwischen auswendig kennen.

Manche Paketquellen erlauben zudem Republishing-Angriffe, die das Paket hinter einer existierenden Version einfach austauschen. Gelegentlich verschwinden zudem Bibliotheken aus den Repositories und brechen damit Builds: Als Folge eines Disputs über die Namensrechte an einem npm-Modul hatte der ursprüngliche Autor 2016 das populäre Paket left-pad zurückgezogen, was zu Hunderten fehlgeschlagenen Builds über mehrere Stunden geführt hat.

Im Fall von Cloud-Anwendungen tritt das Container-Image als finales Produkt an die Stelle des fertig kompilierten Programms. Die genannten Angriffsflächen lassen sich genauso auf die Container-Images übertragen. Das verwendete Basis-Image ist somit nur ein weiterer Zweig im Abhängigkeitsbaum der gesamten Software.

Im Gegensatz zu etablierten Software-Ökosystemen mit gewachsenen Build-Tools wie für Java und JavaScript hält Go als jüngere Programmiersprache mit Blick auf Cloud-Native-Anwendungen für viele Herausforderungen einen passenden Ansatz bereit. Das betrifft unter anderem die Unveränderbarkeit von go.mod in Builds, die Checksum Database oder die statisch gelinkten Binaries als Build-Ergebnis, die sich ideal für Container eignen.

Fixierte Abhängigkeiten

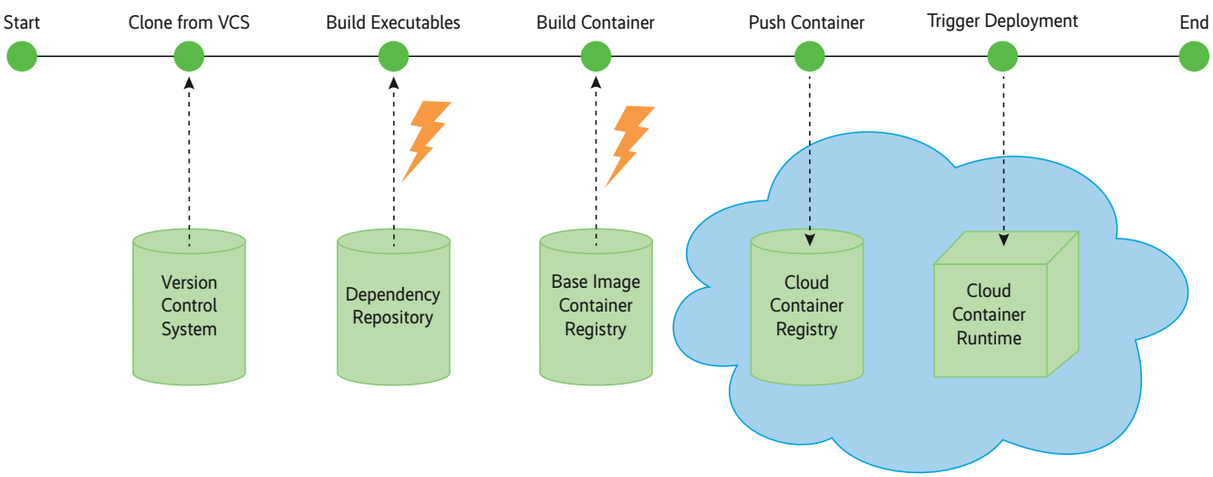
Viele Paketmanager erlauben es, Versionen unvollständig oder dynamisch anzugeben, worauf sie neuere Patchversionen einer Abhängigkeit integrieren. Beispielsweise kennt npm die Syntax `^1.0.0` und `~1.0.0`, die beliebige Minor- beziehungsweise Patch-Versionen erlaubt. Die populären Java-Build-Systeme Maven und Gradle unterstützen ähnlich dynamische Versionsangaben. Pakete können außerdem ihrerseits wiederum dynamische Abhängigkeiten haben. Es ist somit oft unmöglich, die tatsächlich verwendeten Pakete und ihre Versionen vor einem ausgeführten Build zu kennen.

Während dynamische Versionsangaben das Problem automatischer Updates lösen, verhindern sie stabile Builds, da sich die Version bei der nächsten Ausführung auf dem Build-System verändert haben kann. Schlimmer noch: Sie ermöglichen es,

iX-TRACT

- Anders als npm, Gradle und Maven macht der Paketmanager von Go strikte Vorgaben, die viele Supply-Chain-Angriffe vereiteln.
- Dank statisch gelinkter Binaries sind alle Dependencies direkt in der ausführbaren Datei enthalten.
- Go bietet einen guten Schutz für den Einsatz in Containern.

222231344022964431 8.idd 99 11.08.2022 15:44:20



Eine typische CI/CD-Pipeline umfasst mehrere Schritte mit externen Abhängigkeiten (Abb. 1).

vollautomatisch neue Versionen mit Schadsoftware in Programme einzuschleusen.

Ein Ansatz ist eine Lock-Datei, in der bestimmte Versionen für direkte und transitive Abhängigkeiten fixiert sind. Spezielle Befehle wie `npm ci` installieren daraufhin im Gegensatz zu `npm install` nur genau die Versionen aus der Lock-Datei. Leider sind diese Befehle aufgrund ihrer seltenen Verwendung nicht allen bekannt.

Die Entwickler von Go und dessen Paketmanager haben sich gegen eine Lock-Datei entschieden. Die go.mod-Datei enthält die genauen Versionsangaben aller verwendeten Abhängigkeiten, inklusive aller transitiven. Go-Programme haben typischerweise weniger Dependencies als vergleichbare Java- oder JavaScript-Applikationen. Das liegt einerseits an der modernen Standardbibliothek und andererseits daran, dass Go-Entwicklung oft getreu dem Motto „Ein bisschen kopieren ist besser als eine neue Abhängigkeit" läuft. Es gibt somit wenig transitive Abhängigkeiten und die go.mod bleibt übersichtlich.

Enthält die Datei widersprüchliche oder unvollständige Angaben wie `v1` statt `v1.0.2`, schlägt der Build fehl. Seit Go 1.16 verändern außerdem nur die Befehle `go get` und `go mod tidy` die go.mod. Die Konsequenz ist, dass die Datei die vollständige Wahrheit über alle verwendeten Abhängigkeiten enthält, die dadurch im Versionskontrollsystem ersichtlich sind. Bei Java hingegen muss der Paketmanager zunächst den Abhängigkeitsbaum berechnen.

Statische Dependency-Scans

Die Abhängigkeiten zu anderer Software und der Umstand, dass in nahezu jeder Software im Laufe der Zeit Schwachstellen entdeckt werden, macht eine regelmäßige Kontrolle und Updates der Dependencies zu einem Pflichtthema. Die Erfahrung zeigt, dass eine manuelle Kontrolle nicht genügt, sondern ein automatischer Prozess die Dependencies auf Schwachstellen untersuchen muss. Idealerweise findet die Prüfung bei jedem Erstellen auf dem Build-System statt, um frühzeitig frisch in den Abhängigkeiten entdeckte Sicherheitslücken zu erkennen.

Node.js-Anwendungen prüft der Befehl `npm audit` auf Schwachstellen. Als Quelle dient die Lock-Datei. Der Befehl kann versuchen, die Probleme direkt zu beheben. Für Java-Anwendungen empfiehlt sich der OWASP Dependency Check, für den Plug-ins und ein Kommandozeilenprogramm für die populärsten Build-Systeme existieren. Er lässt sich ebenso für Go-Programme nutzen. Momentan muss dafür noch der experimentelle Modus aktiviert werden:

```
dependency-check \
  --enableExperimental \
  --scan go.mod
```

Als Alternative gibt es das Tool Nancy, das die Ausgabe von `go list` im JSON-Format per `stdin` annimmt:

```
go list -json -deps | nancy sleuth
```

Wer GitHub nutzt, kann den Dependabot oder Renovatebot aktivieren, die für viele Programmiersprachen und Build-Systeme die Abhängigkeiten analysieren und bei Schwachstellen die Repository-Besitzer alarmieren. Teilweise erlauben die Services sogar automatisierte Pull Requests für Updates.

Alle genannten Verfahren führen jedoch nur eine statische Prüfung aufgrund der Paketnamen und deren Versionsnummern durch. Die Tests erfolgen direkt auf den eingecheckten Dateien, und teilweise ist zunächst eine Auflösung des Abhängigkeitsbaums erforderlich. Dass die Dienste den Inhalt der Pakete nicht prüfen, öffnet den Raum für Republishing-Angriffe.

Vertrauenswürdige Abhängigkeiten

Ein schwerwiegendes Problem bei Dependencies sind Republishing-Attacken: Angreifer veröffentlichen unter der gleichen Versionsnummer ein anderes Paket. Viele Paketmanager legen zwar Prüfsummen der Pakete in den Lock-Dateien ab. Das verhindert den Republishing-Angriff, der nicht nur über die Versionsnummer kommen kann, wenn auch nur bei bekannten Dependencies. Mittlerweile sind private npm Scopes ein beliebtes Angriffsziel. Sie ermöglichen es, Pakete unter einem Präfix wie `@mycorp` zusammenzufassen.

Für Scopes lässt sich eine alternative Registry festlegen, was besonders für Firmen interessant ist, die ihre eigenen Pakete in einer firmeninternen Registry veröffentlichen. Normalerweise erfolgt die Angabe der Registry für einen Scope über `npm config` oder über die .npmrc-Datei. Fehlen die Angaben, fragt npm standardmäßig npmjs.com an. Angreifer legen nun einen Account mit genau diesem Scope auf npmjs.com an, die Pakete

222231344022964431 8.idd 100 11.08.2022 15:44:20

mit Malware enthalten. Da Rechner von Entwicklern fast immer Internetzugriff haben, kann das falsche Repo auch in die Lock-Datei geschrieben werden.

Um Republishing-Angriffe abzuwehren, bieten sich unterschiedliche Methoden an, die sich in ihrer Effektivität unterscheiden: Konsequentes Prüfen von Hashes verhindert nicht nur fehlerhafte Pakete, sondern deckt eventuell zusätzlich veränderte Pakete auf, wenn ein Vergleich mit bekannten Prüfsummen möglich ist. Noch eine Stufe stärker sind kryptografische Signaturen, die jedoch die wenigsten Paketmanager verarbeiten, zudem sind nur wenige Pakete signiert. Eine weitere Schwierigkeit: Ohne ein Web of Trust müssen Teams darauf vertrauen, dass der gespeicherte öffentliche Schlüssel tatsächlich von dem Originalautor oder der Originalautorin stammt.

Gradle kann seit einiger Zeit sowohl Prüfsummen als auch Signaturen von Dependencies überprüfen. Maven prüft Checksummen und kann PGP-Signaturen über ein Plugin verifizieren. npm erlaubt es, über einige Schritte die Integrität von heruntergeladenen Paketen über PGP zu überprüfen.

Als praktische Maßnahme gegen Republishing bietet sich noch ein eigenes Proxy Repository an. Als positiver Nebeneffekt können die Build-Systeme vom Internet isoliert arbeiten, womit sie vor Angriffen geschützt sind.

Go vermeidet Republishing teilweise durch eine zentrale Datenbank mit Prüfsummen, die Checksum Database oder kurz sumdb. Sie enthält die Hashes aller jemals veröffentlichten Versionen von Go-Dependencies in Form eines Hash-Baums unter sum.golang.org. Auf ihm lässt sich im Gegensatz zu einer regulären Datenbank die Integrität der Inhalte ähnlich wie bei einer Blockchain verifizieren, auch wenn man letztlich Google als Betreiber vertrauen muss. Die Go-Toolchain ruft die Prüfsummen aller Dependencies aus der Datenbank ab und vergleicht sie mit denen der heruntergeladenen Dependencies. Sollte eine Prüfsumme nicht übereinstimmen, schlägt der Build fehl. Falls eine Version einer Dependency noch nicht in der Datenbank ist, ruft die Datenbank die Dependency ab, bildet die Prüfsumme und fügt sie dem Hash-Baum hinzu. Danach kann die Prüfsumme einer Version nicht mehr verändert werden.

Weniger zu den Gefahren als in die Kategorie der Unannehmlichkeit fällt das versehentliche oder absichtliche Löschen von Abhängigkeiten. Beliebte Pakete stellt die Community normalerweise schnell wieder her, aber bei seltenen Packages hilft nur ein eigenes Proxy Repository, das Kopien der Dependencies bereithält. Go hat im Gegensatz zu anderen Ökosystemen kein zentrales Repository, bietet jedoch über proxy.golang.org einen Module Mirror an, der Kopien der wichtigsten Pakete bereitstellt.

Einige Paketmanager wie npm, Gradle und Maven erlauben Post-Install-Hooks, also das Ausführen von Code aus der Dependency direkt nach der Installation. Was einerseits nützlich sein kann, ermöglicht andererseits die vollautomatische Infektion eines Rechners mit Schadsoftware, wenn das Projekt eine manipulierte Abhängigkeit hat. Bei unbekannten Quellen empfiehlt es sich, npm mit dem Flag `--ignore-scripts` aufzurufen. Plugins für Build-Systeme wie Maven oder Gradle sind ein ähnliches Einfallstor, da sie wie Dependencies aus einem Repository heruntergeladen werden. Go führt weder beim Installieren noch beim Kompilieren Code aus heruntergeladenen Dependencies aus. Andererseits wird eine im Code verbaute Abhängigkeit spätestens mit Start des Programms ohnehin ausgeführt.

Container und Images

Eine Cloud-Native-Applikation wird üblicherweise als Container ausgeliefert, der die Anwendung und ihre Abhängigkeiten bündelt. Im Fall von Java sind das typischerweise die eigentliche Anwendung, ihre Java-Bibliotheken und eine Laufzeitumgebung für Java. Hinzu kommen oft Hilfsprogramme zum Überwachen der Applikation sowie zum Debuggen. In vielen Containern schlummern sogar ganze Linux-Distributionen mit allen gängigen Tools wie einem Paketmanager zum Nachinstallieren von Software. Dazu kommt, dass die Pods im Kubernetes oftmals auch noch Internetzugriff haben, womit beliebige Software nachgeladen werden kann. Die in einem Container enthaltene Software kann genau wie die Anwendung selbst Schwachstellen enthalten. Da es üblich ist, vorgefertigte, unspezifische Basis-Images nur um die Anwendung und ihre Abhängigkeiten zu erweitern, ist hier die Gefahr von veralteter Software besonders groß. Im stressigen, von der Feature-Entwicklung getriebenen Alltag werden außerdem regelmäßige Updates des Basis-Images gerne vergessen. Die im Basis-Image verbauten Abhängigkeiten lassen sich nur mit speziellen Tools wie dive oder durch Analyse der Container-Quelltexte sichtbar machen.

Um die Angriffsfläche gering zu halten, empfiehlt es sich, die Abhängigkeiten im Image auf das benötigte Minimum zu reduzieren. Go hat im Vergleich zu JavaScript oder Java den Vorteil, dass es weder dynamisch kompiliert werden muss noch eine virtuelle Maschine zum Ausführen benötigt. Da Go weitgehend auf statisch gelinkte Binaries setzt, sind alle Abhängigkeiten im Executable enthalten. Es gibt keine zusätzlichen Dependencies zu Systembibliotheken. Ein einfaches Go-Programm lässt sich problemlos in einem komplett leeren Container ausliefern:

```
FROM scratch
COPY myapp /myapp
ENTRYPOINT ["/myapp"]
```

Ein solcher Container enthält nur die Anwendung ohne eine Shell, einen Paketmanager oder andere Anwendungen, die in anderen Container-Images enthalten sind. Selbst wenn die Anwendung eine Sicherheitslücke enthält, die das Ausführen einer Shell ermöglicht, läuft der Angriff ins Leere: Es gibt schlichtweg keine Kommandozeile. Ein weiterer Vorteil ist die geringe Größe des Images – es ist genauso groß wie die Anwendung selbst. Der Preis für die Sicherheit ist ein Verlust an Bequemlichkeit: Es ist nicht möglich, sich mit einem laufenden Container zu verbinden und Befehle auszuführen. Bei Bedarf kann ein Ephemeral Container helfen, also ein kurzlebiger Debug-Container wie BusyBox, der im selben Pod läuft. Dauerhaftere Abhilfe schafft ein Sidecar-Container oder ein separater Service.

Für Businessanwendungen kann der leere Container zu spärlich sein, und häufig benötigen die Applikationen eine Liste vertrauenswürdiger Stammzertifikate. Ein bewährtes Vorgehen ist, die Anwendung ohne Root-Rechte auszuführen. Dafür bietet sich der Einsatz eines Google Distroless Image an. Das Image gcr.io/distroless/static enthält lediglich einen Standardsatz vertrauenswürdiger Stammzertifikate, einen Eintrag in /etc/passwd für den Root-User, ein /tmp-Verzeichnis und tzdata für Zeitzonenunterstützung. Die zusätzlichen Daten erzeugen im Image lediglich einen Overhead von ungefähr 2 MB. Im Dockerfile ändert sich nur die erste Zeile:

```
FROM gcr.io/distroless/static
COPY myapp /myapp
ENTRYPOINT ["/myapp"]
```

Für einen unkomplizierten Einstieg in Go-Container für einfache Anwendungen bietet sich das Projekt ko an. Es baut die Anwendung lokal, verpackt sie in einen gcr.io/distroless/static-Container und kann das fertige Image in eine Container Registry pushen. Es erfordert keine Docker-Installation.

222231344022964431 8.idd 101 11.08.2022 15:44:20

Kontinuierliche Kontrolle

Wer zusätzlich glibc benötigt, weil die Anwendung beispielsweise eine Dependency auf eine C-Bibliothek über Cgo hat, kann das Image gcr.io/distroless/base verwenden. Daneben existieren passende Distroless-Images für Java, Python, Node.js und Rust.

Für plattformunabhängige, reproduzierbare Go-Builds kann sich ein Blick auf Multi-Stage-Dockerfiles lohnen. Dabei wird der Quellcode in einen Container wie golang:1.18 kopiert und dort kompiliert. Das fertige Binary lässt sich direkt vom Build-Container in den Runtime-Container kopieren. Auf dem ausführenden System muss nur Docker installiert sein. Damit entfällt die Installation von Go und der Code kompiliert konsistent mit derselben Go-Version.

Statische Container-Scans

Wer ein nicht triviales Basis-Image für die eigene Anwendung wählt, sollte das fertige Image und die regulären Abhängigkeiten automatisiert auf Schwachstellen überprüfen lassen. Das geschieht idealerweise im Rahmen der CI/CD-Pipeline direkt nach dem Build des Images. Die Analyse erfolgt statisch, ohne das Image auszuführen. Erkennt das Tool eine Schwachstelle, bricht es den Build ab oder verschickt zumindest eine Benachrichtigung. Es empfiehlt sich, fortlaufend während und nach der Entwicklungsphase auf Schwachstellen zu scannen, um gegebenenfalls schnell auf eine neu aufgetretene Sicherheitslücke reagieren zu können.

Die optimale Prüfung auf Schwachstellen hängt von der jeweiligen Infrastruktur ab. Bei den großen Cloud-Anbietern gehören Scans der Container-Images zum Leistungsumfang. Je nach Einstellung scannen sie Images direkt beim Push in die jeweilige Container Registry automatisch. Teams müssen sich lediglich überlegen, wie sie die Ergebnisse in die CI/CD-Pipeline integrieren. Für GitHub existieren zahlreiche Actions. Im Zusammenspiel mit Jenkins und anderen Build-Systemen bieten sich die Kommandozeilentools der Cloud-Anbieter an. Da der Push des Images ohnehin von der Kommandozeile erfolgt, ist der Abruf des Ergebnisses normalerweise nur ein weiterer Befehl. Am Beispiel von AWS sieht das folgendermaßen aus:

```
aws ecr start-image-scan --repository-name myapp \
  --image-id imageTag=1.0.0
aws ecr describe-image-scan-findings \
  --repository-name myapp \
  --image-id imageTag=1.0.0
```

Ist Docker Desktop installiert, kann der Scan mit

```
docker scan myapp:1.0.0
```

erfolgen, wobei im Hintergrund ein Drittanbieterdienst läuft. Die Anzahl der Scans pro Nutzer und Monat hängt von der Lizenz ab. Wer auf private Container-Registries im eigenen Rechenzentrum festgelegt ist, kann einen Blick auf die Projekte Trivy und Clair werfen. Alternativ gibt es einige kommerzielle Anbieter, die sich auf das Prüfen von Abhängigkeiten und Schwachstellen spezialisiert haben und mit Besonderheiten wie automatischen Upgrades der Basis-Images werben.

222231344022964431 8.idd 102 11.08.2022 15:44:20

Listing 1: Signierte Images

```
$ docker trust inspect --pretty nginx:1.23.0       # Shows trust data
$ DOCKER_CONTENT_TRUST=1 docker pull nginx:1.23.0  # Successfully pulls the image
$ DOCKER_CONTENT_TRUST=1 docker pull curlimages/curl:7.83.1
Error: remote trust data does not exist [...]
$ DOCKER_CONTENT_TRUST=1 docker pull gcr.io/distroless/static:latest
Error: error contacting notary server [...]
```

Nach dem Deployment sollten Teams die verwendeten Images weiter auf neue Schwachstellen prüfen. Das gilt besonders für Images wie nginx, prometheus oder traefik, die man typischerweise nicht baut, sondern einfach ausführt. Für kontinuierliche statische Scans der laufenden Container-Images bietet sich der Trivy-Operator an. Nach der Installation im Cluster als Custom Resource Definition (CRD) scannt er automatisch alle sechs Stunden die laufenden Images. Das Ergebnis legt er als eigene Ressource im Cluster ab, die Weiterverarbeitung beziehungsweise Visualisierung kann mit Prometheus oder der Lens Extension erfolgen.

Dynamische Scans, also Scans der Software in den laufenden Instanzen, sind zumindest momentan kein Thema. Prinzipiell ist zwar denkbar, dass über eine Sicherheitslücke Schadsoftware in einen laufenden Pod gelangt, aber der Einsatz eines stark reduzierten Basis-Images wie Distroless lässt dafür wenig Spielraum.

Vertrauenswürdige Container-Images

Container-Images sind an sich unveränderbar. Für deren Tags und damit die Versionsnummern gilt das jedoch nicht. Prinzipiell ist es möglich, ein Tag wie `latest` auf ein anderes Image zu kleben. Typischerweise speichern Systeme die Container in der lokalen Container Registry zwischen und fragen eine geladene Version nicht erneut an. Verschwindet der Container jedoch aus der Registry – beispielsweise durch das Leeren des Cache oder weil ein neuer Rechner oder Build-Knoten hinzukommt – kann er sich zwischenzeitlich verändert haben. Um solchen Republishing-Angriffen zu begegnen, lässt sich ein bestimmtes Image über einen Digest referenzieren. Dabei handelt es sich um die SHA-256-Prüfsumme des Images, die beim Ändern des Inhalts einen anderen Wert hat.

Wer zusätzlich sicher sein will, dass das Image von den Entwicklerinnen und Entwicklern eines Projekts stammt, kommt nicht um kryptographisch signierte Images herum. Dafür gibt es die konkurrierenden Projekte Notary / Docker Content Trust und Sigstore Cosign. Ersteres ist in Docker Desktop integriert und lässt sich mit einer Umgebungsvariable aktivieren. Danach erlaubt Docker Desktop nur noch das Pullen und Ausführen signierter Images (Listing 1).

Sigstore Cosign (kurz für Container Sign) erlaubt unabhängig von Docker Desktop das Validieren von Images über das separate Tool `cosign`. Zusätzlich ist der öffentliche Schlüssel des Unterzeichners erforderlich:

```
$ cosign verify --key distroless-cosign.pub \
  gcr.io/distroless/static:latest
```

Umgekehrt lassen sich bei beiden Projekten die eigenen Images signieren. Kubernetes kann anschließend die Images vor dem Deployment über einen `ValidatingWebHook` validieren und somit nur signierte Images ausführen.

Geschützte Lieferkette in Go

Zusammenarbeit und Open-Source-Software hat die Softwareentwicklung in den letzten Jahren beflügelt. Dennoch birgt jede Abhängigkeit Risiken. Immer wieder werden neue Schwachstellen sogar in weitverbreiteten Paketen gefunden. Längst haben Kriminelle erkannt, dass Dependencies ein Weg in Firmennetze und auf Server sind. Im Unternehmensumfeld steigt das Bewusstsein für Sicherheit in der Lieferkette: Das Framework SLSA (Supply Chain Levels for Software Artifacts) beschreibt ein Reifegradmodell mit mehreren Stufen der Integrität für Projekte und Unternehmen. Es bietet Anleitungen und Checklisten, ohne sich auf bestimmte Tools oder Techniken zu begrenzen.

Der beste Schutz ist, die Anzahl der Dependencies gering zu halten. Go erlaubt es dank der gut ausgestatteten Standardbibliothek und flachen Abhängigkeitsbäumen, komplexe Programme mit wenigen Abhängigkeiten zu schreiben. Noch besser gelingt das bei Containern, da die statisch gelinkten Binaries mit fast leeren Basis-Images auskommen.

Gegen Schwachstellen in Abhängigkeiten helfen nur automatisierte Scans und regelmäßige Updates. Da es in Go keine versteckten oder transitiven Abhängigkeiten gibt und die go.mod die exakten Versionen aller Dependencies enthält, genügt eine statische Analyse der go.mod-Datei. Bei Basis-Images wie Scratch oder Distroless erübrigt sich mitunter sogar der Scan der Container-Images.

Die Go Checksum Database und der Verzicht auf Abhängigkeiten mit dynamischen Versionsangaben erschweren Republishing-Angriffe und verhindern das Einschleusen von Schadsoftware durch automatische Dependency-Updates. Andere Build-Systeme erfordern meist deutlich mehr Maßnahmen, um die gleiche Stufe an Sicherheit zu erreichen. Einen Mehrwert kann unter Abwägung der Kosten ein kontrolliertes Proxy-Repository bringen. Künftig werden kryptografische Signaturen wohl eine wachsende Rolle beim Build einnehmen.

(rme@ix.de)

Quellen

Die Links zu den Tools, Berichten über die Schwachstellen und weiteren Ressourcen finden sich unter ix.de/zzdn.

Bernhard Saumweber

ist leidenschaftlicher Entwickler und Software Architect bei der QAware GmbH. Neben User Experience und sicheren Cloud-Native-Anwendungen begeistert er sich für innovative Programmiersprachen und gibt sein Wissen darüber in Vorlesungen und Vorträgen weiter.

222231344022964431 8.idd 103 11.08.2022 15:44:20