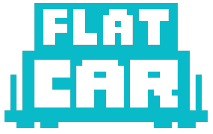


Flatcar @ TAG Security

Technical Overview

Motivation & Goal

- Flatcar is a complex project
  - Provide technical overview
  - Introduction to concepts and processes
- Gain understanding
  - Foundation for elaborate questions
  - Basis for determining next steps

Agenda

Background, Concepts, and Operation

- History, Basic Concepts, FS Layout & Mount Points
- Using Flatcar: Configuration, Provisioning, Operation
- Update Process
- Test, Release, and Stabilisation

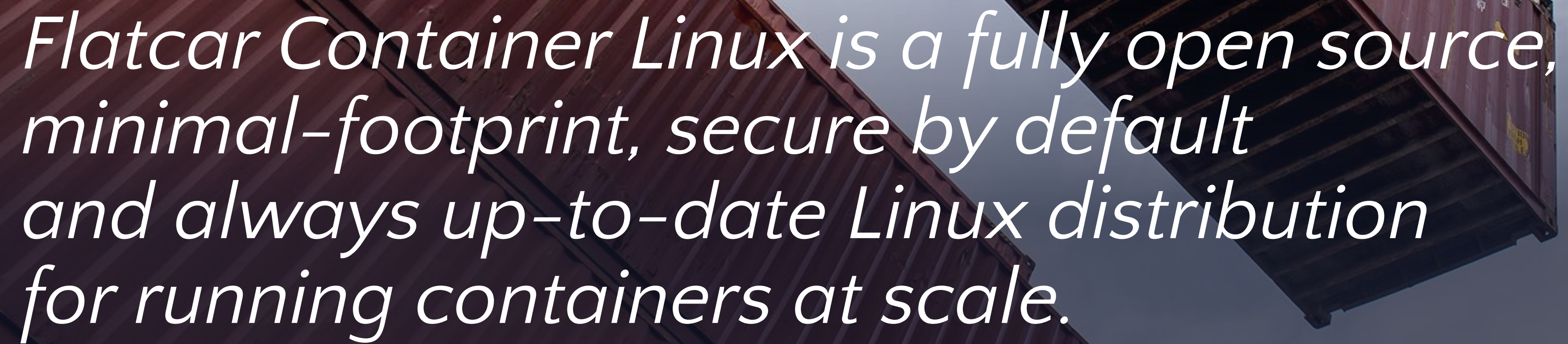
Developing Flatcar

- Packages and Concepts
- Reproducible Builds

Scope limits

Security Considerations Wrap-Up

Appendix // SLSA deep dive, Vendor support



Flatcar Container Linux is a fully open source, minimal-footprint, secure by default and always up-to-date Linux distribution for running containers at scale.

[Flatcar Container Linux project Mission Statement]

Secure by Default

Minimal attack surface (small footprint with very few run-time services)

Securely operate workloads, secure update process

Secure Build and Release

“Security First” Development Process

Background, Concepts, and Operation

Background, Concepts, and Operation

Heritage

Flatcar Container Linux started as a Friendly Fork of the epochal
CoreOS Container Linux, which in turn was derived from
Chromium OS, which is based on
Gentoo Linux.

Flatcar's core concepts are derived from this heritage.

We always build from sources
like Gentoo does.

The OS is
immutable and shipped as a full disk image
uses A/B partitioning for updates (and rollbacks), distributed via a stateful protocol (Omaha)
like Chromium OS.

Flatcar includes
a minimal set of applications and tools tailored to only run containers
it uses declarative configuration applied at provisioning time
like CoreOS.

Our SDK makes development accessible, and is a key element for SLSA. Automation helps with Freshness and security.
(This has been a major area of investment for us.)

Background, Concepts, and Operation

Disk Layout and mount points

- Initial provisioning uses full disk images.
- Updates contain only kernel+initrd+USR
- Additional disks/storage customised / configured via Ignition.

#	label	mountpoint	size	type
1	EFI SYSTEM	/boot	128M	EFI
2	BIOS_BOOT		2M	BIOS boot
3	USR-A	/usr[1]	1G	ext2 (ro)
4	USR-B	/usr[1]	1G	ext2 (ro)
5	ROOT-C		0	*<reserved>*
6	OEM	/oem[2]	128M	BTRFS
7	OEM-CONFIG	vendor specific	64M	vendor specific
8	<reserved>		0	<reserved>
9	ROOT	/	what's left[3]	ext4, BTRFS, XFS

[1] A/B OS partitions. Only one is active. P4 is empty when initially provisioned.
[2] Used to be /usr/share/oem; move to /oem w/ Alpha next week.
[3] Partition is expanded at provisioning time to entire remaining disk.

1. + 2. EFI partition with kernel+initrd, BIOS boot partition for legacy boot
`/boot` : EFI partition. (kernel+initrd)s for use with USR-A / USR-B

3. + 4. Read-only OS partitions.
`/usr` : OS binaries. Read-only, dm-verity protected partition.
One is active, the other used to stage update.

6. + 7. OEM tooling for vendor support. Specific to each vendor image.
`/usr/share/oem`: E.g. openvm-tools (vmware), ssm-agent (AWS)

9. Root: formatted + populated at first boot, expanded to the end of disk.
`/` : user configuration / customisation, container images etc.

`/etc` : overlayfs backed by `/usr/share/flatcar/etc` (lowerdir)
Allows for for user/vendor configuration management

`/dev/shm`, `/run`, `/media`: tmpfs; `/dev`: devtmpfs

Background, Concepts, and Operation

Provisioning and Boot

Before provisioning / first boot: Configuration

- Butane (YAML) / Ignition (JSON). Declarative. Applied ***once*** (at provisioning time).
- Includes full host configuration (though basic settings are auto-detected).
  - set up disks and filesystems; create directories and files (inline or download).
  - Create and manage systemd units. Can also be used for per-boot configuration.
  - Create users and groups, deploy SSH keys (authorized_keys).
  - Configure kernel arguments

Boot process

1. Grub: Load kernel + initrd for "active" USR partition. One (kernel+initrd+USR) per installed OS version.
   a. Alternatively, Grub one-time boots "new" configuration (first boot after update)
2. Kernel + initrd: mount root and `/usr`, mount `/etc` overlay.
   a. At first boot, expand, format, and populate `/` , apply Ignition configuration.
   b. `/usr` partition dm-verity is built into initrd.
   c. Pivot root `/sysroot -> /`
3. Systemd reloads; regular user space starts up

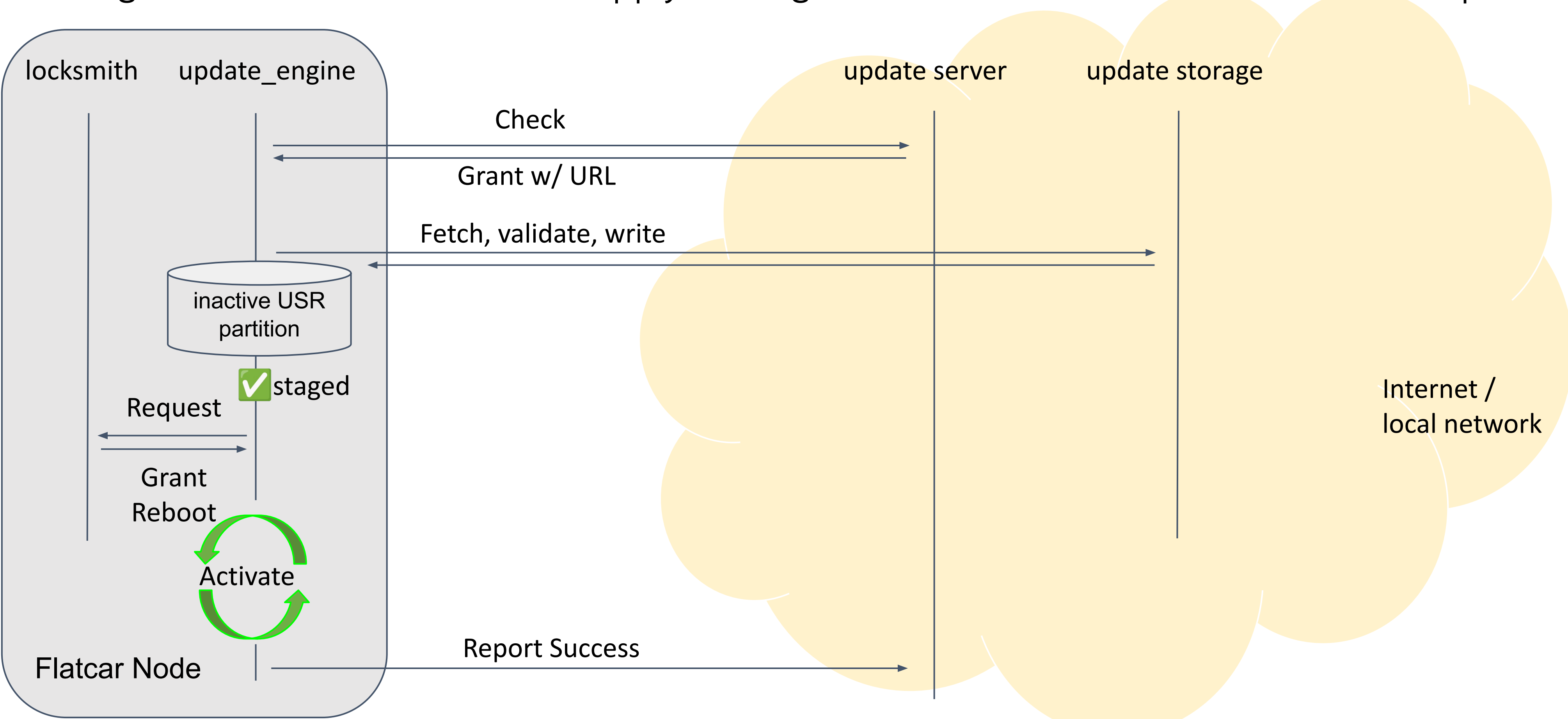
Background, Concepts, and Operation

Staying up to date

Flatcar inherits the Omaha protocol and A/B partition atomic updates / roll-backs from its Chromium OS roots.

Update payload is a signed binary containing an image of the /usr partition + kernel + initrd.

Updates are activated through reboot. Different schemes apply for single nodes, clusters with / without control plane.



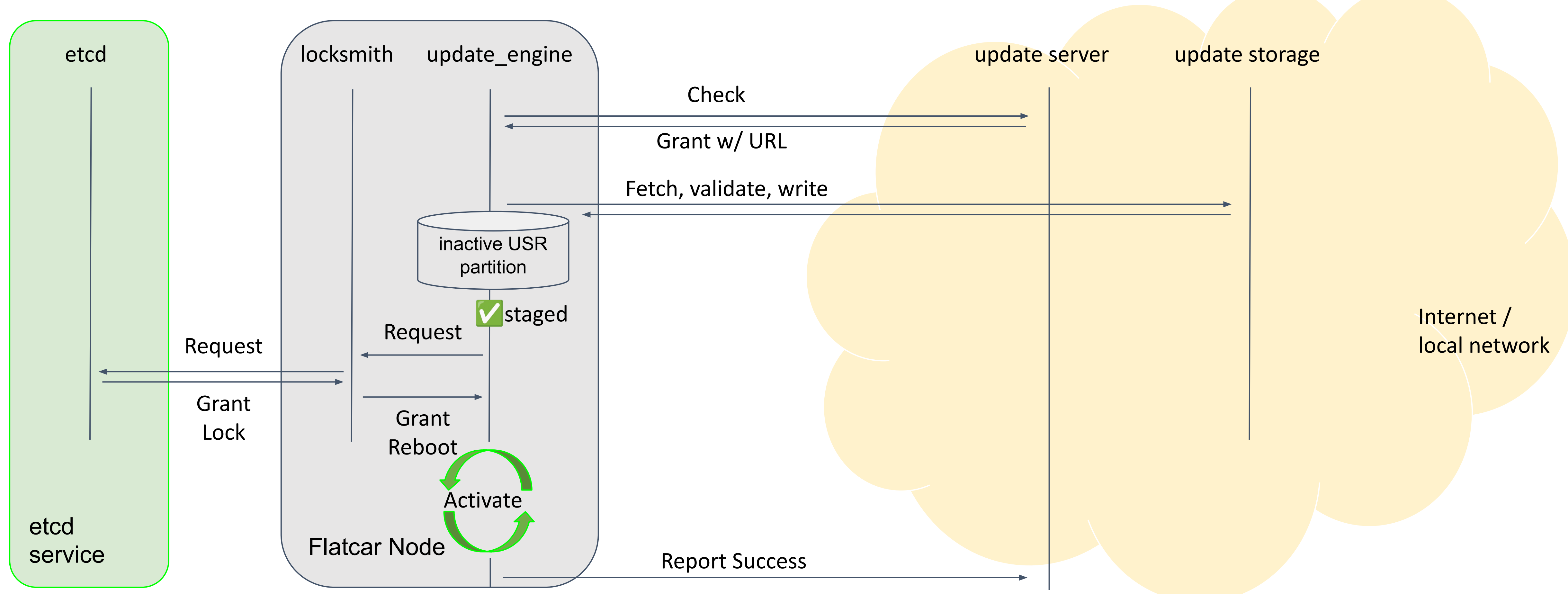
Background, Concepts, and Operation

Staying up to date

Flatcar inherits the Omaha protocol and A/B partition atomic updates / roll-backs from its Chromium OS roots.

Update payload is a signed binary containing an image of the /usr partition + kernel + initrd.

Updates are activated through reboot. Different schemes apply for single nodes, clusters with / without control plane.



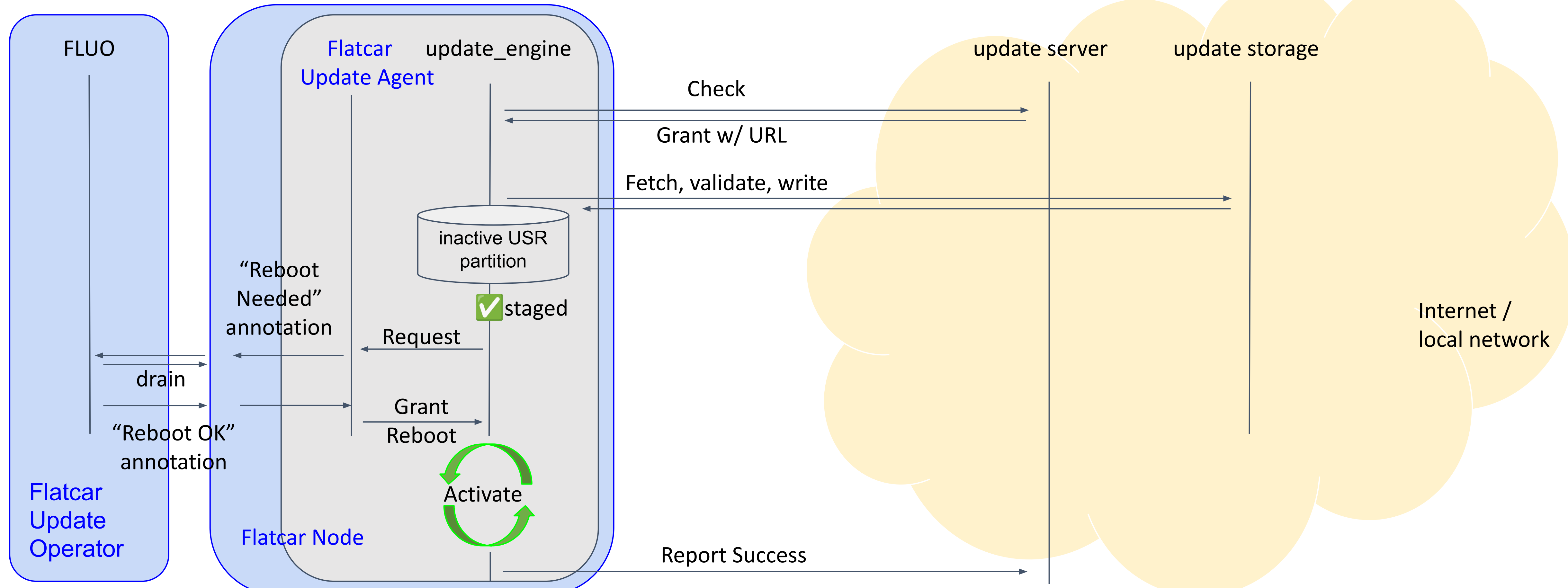
Background, Concepts, and Operation

Staying up to date

Flatcar inherits the Omaha protocol and A/B partition atomic updates / roll-backs from its Chromium OS roots.

Update payload is a signed binary containing an image of the /usr partition + kernel + initrd.

Updates are activated through reboot. Different schemes apply for single nodes, clusters with / without control plane.

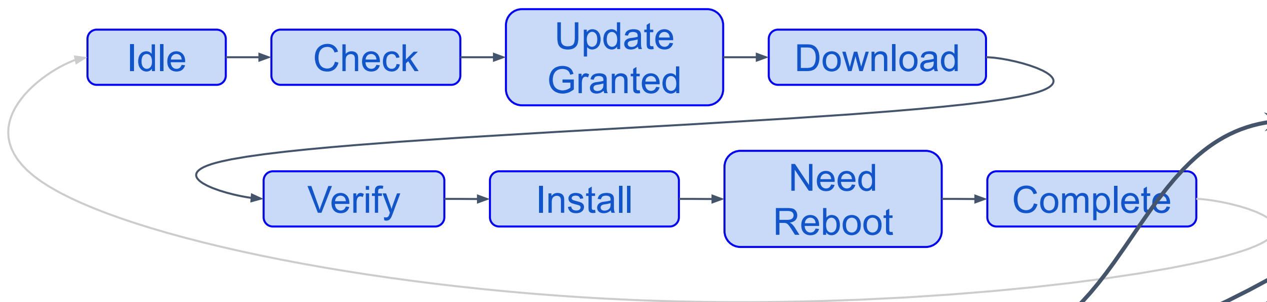


Background, Concepts, and Operation

Nebraska Update Server

FOSS update server

- part of the Flatcar Container Linux project (generic Omaha implementation, not Flatcar specific)
- Stateful update protocol



- Insights into OS version spread, update uptake, Client status break-down / errors
- Server supports rate limiting / staggered rollout, custom instance groups, etc.

Stable (AMD64)

stable

For production clusters (AMD64)

CHANNEL

S stable 3510.2.2 (AMD64) 06/01/2023

UPDATES

Enabled ✓

ONLY OFFICE HOURS

No

SAFE MODE

No

UPDATES POLICY

Unlimited number of parallel updates

UPDATES TIMEOUT

1 days

Update Progress 30 days · 7 days · 1 day · 1 hour

INSTANCES

44884

See all instances

39.8% Complete
56.7% Downloaded
0.0% Other
0.0% Installed
1.7% Downloading
0.7% Error

Version Breakdown 30 days · 7 days · 1 day · 1 hour

60000
45000
30000
15000
0
12:00 18:00 14 Jun 06:00

Showing data for the last time point.
Click the chart to choose a different time point.

Version	Count	Percentage
3510.2.2	18141	40.4
3374.2.4	9076	20.2
3510.2.1	4485	10.0
3374.2.5	2016	4.5
3033.2.3	1242	2.8

Rows per page: 5 1-5 of 64

Status Breakdown 30 days · 7 days · 1 day · 1 hour

3200
2400
1600
800
0
12:00 18:00 14 Jun 06:00

Showing data for the last time point.
Click the chart to choose a different time point.

Status	Version	Instances
Update Granted	3510.2.2	605
Downloading	3510.2.2	587
Downloaded	3510.2.2	375
Error	3510.2.2	199
Complete	3510.2.2	15

Rows per page: 5 1-5 of 8

Background, Concepts, and Operation

Nebraska Update Server

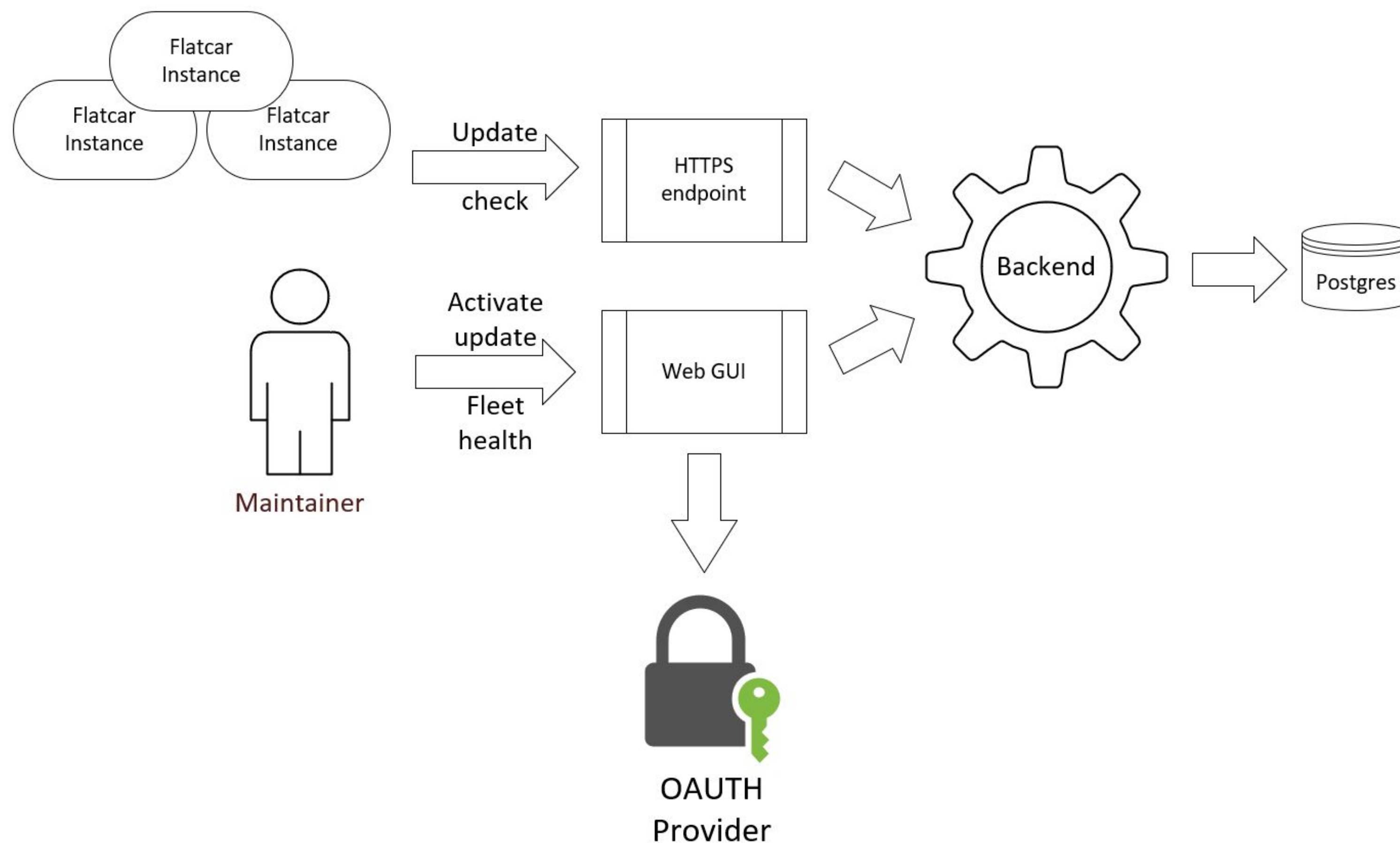
Serves Metadata,
  not actual payloads

Written in Go, TypeScript

Instance access is not authenticated

Web GUI authenticated via OAuth
  (Flatcar maintainers team)

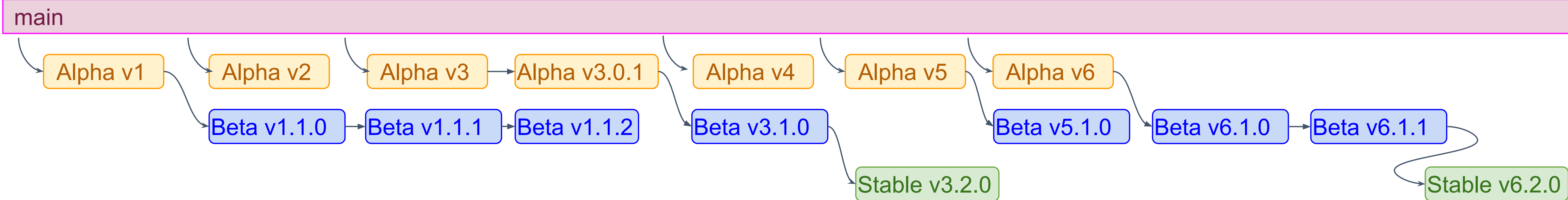
Runs on AWS RDS w/ ELB
  for basic DOS protection



Background, Concepts, and Operation

Test, Release, and Stabilisation Process

Stabilisation through Channel promotion: Alpha, Beta, Stable [, LTS]



Alpha is for development, Beta for Canaries, Stable for Prod.
ALL releases must pass full release test suite.

Releases are built on private infrastructure hosted on Equinix Metal (deep dive see SLSA appendix)

- Access to infrastructure is restricted to a subset of the Flatcar Maintainers team
- Release images (for new provisionings) are signed on build infrastructure
- Update payloads are signed manually by a Flatcar Maintainer in an air-gapped environment
  - Uses a separate signing key (HW token), access is restricted to 3 core maintainers

Developing Flatcar

Packages, concepts,
reproducible builds

Developing Flatcar

Packages and Projects

In total, the project uses less than 500 software packages (Gentoo or self-maintained ebuilds[1])

- 299 packages make up the Base OS Image
- 355 packages are used for the Flatcar developer container (most shared with the base OS).
- 412 packages are used in the SDK (most shared with the two above).

Most packages[1] we use are from Gentoo upstream, but we also maintain some ourselves.

- We sync updates to most packages from Gentoo upstream, including security fixes
- We collaborate with upstream Gentoo on fixes, stabilisations, and package updates
- We maintain our own ebuilds[1] for Flatcar features, and for packages where we deviate from Gentoo
  - important OS components (kernel and systemd): track upstream releases directly independent from Gentoo
  - also Flatcar specific tools like Ignition, Afterburn, or Locksmith
  - These we update based on the respective upstream project's releases, usually up to date

Nebraska

- No major new features planned at this time
- Low intensity maintenance mode (keeping libs & deps up to date)

Flatcar Linux Update Operator (FLUO)

- Kubernetes operator for cluster-wide update coordination; also no major new features planned
- CAPI expressed interest in extending it to also cover on-node Kubernetes

[1] Gentoo does not ship packages like traditional distros do. Instead, build instructions are shipped to build software directly from upstream sources. These instructions are called "***ebuild*** files".

Developing Flatcar

Core Development Concepts

There are two components to Flatcar distro development; the **SDK container** and the **“scripts” repository**. (“scripts” is a legacy name; “Flatcar Distro” would be a more descriptive name for the repository)
“scripts” contains all package definitions (ebuilds) as well as automation to build OS images and the SDK.
“scripts” also contains versioning information and helper scripts to work with the SDK.

Flatcar image builds are *always* from sources[1]. Build result is always an image. There are no packages. SLSA provenance for all OS image packages is recorded during build and included in the OS image.

The Flatcar SDK container offers a full-featured self-contained build environment. It is also used for all automated / CI builds. It is distro independent and can even run on WSL.

Versioning is based on GIT branches and tags in the “scripts” repository.

- Each active maintenance branch is … a branch.
- Each release is a tag
- The “main” branch may be considered “alpha-next”. New major versions are branched off from there.
- Branches are self-contained and include all information to run a build. No parameters necessary.
  Version information is updated / recorded when creating branches and tags.
  A check-out of a specific “scripts” OS version tag or branch will include the respective version information.

[1] We provide pre-built packages for all releases via our packages caches. These are meant to cut down rebuild time of e.g. downstream modifications of Flatcar.

Developing Flatcar

Flatcar builds are reproducible[1]

```
git clone https://github.com/flatcar/scripts.git; cd scripts
git checkout alpha-3619.0.0
./run_sdk_container -t
$ ./build_packages
$ ./build_image
$ ./image_to_vm.sh —from __build__/images/images/amd64-usr/latest/ --format ami
```

will reproduce the AMI image of release alpha-3619.0.0 with the exact package versions, SDK version and configurations used to originally build that release.

Running our test suite with more than 100 scenario tests is equally straightforward as it is also shipped in a self-contained container. For each release, the version of the test suite used to run release tests is also recorded in the scripts repo. Re-running release tests on re-built releases will therefore the exact version used for testing that respective release.

[1] Note that while compilation results are the same, binaries are not bit-by-bit identical because compilers insert transient information like build hosts at compile time.

Security Considerations

Security Processes and CVEs

The Flatcar project takes a proactive approach to security. We maintain a Security Task force to track emerging and to tend to ongoing issues.

- The task force is formed from Maintainers, who can volunteer and are elected into the task force.
- The task force has access to embargoed security issues; access is restricted on a need-to-know basis.
- Our task force also includes an “on-call” Primary and Secondary - roles rotated on a weekly basis.
  - Primary and secondary will actively process incoming security notifications.
  - Once known, issues will be assessed against Flatcars existing mitigations and added to the tracker.
  - They are expected to spend at least a few hours a week actively researching issues in upstream projects, Gentoo GLSAs, and other distros’ security trackers.
- The task force also reviews the state of ongoing issues in a fortnightly cadence, in a private video call.

In PRs that fix a security issue and / or update a package to a release that contains one or more CVE fixes, the CVEs fixed are noted. This is picked up by release notes generation later.

For each release, “CVEs fixed” is part of the release notes. There is a separate detailed report for each release elaborating on each of the CVEs fixed.

Flatcar is shipped as an image; even patch level releases that fix a single issue require a full from-scratch build and thorough testing of all supported vendor platforms. Our average round-trip time from “patch available” to “release published” is between 24h and 48h.

Scope Limits

Separation of Concern

Separation of Concern and Scope Limits

The Isolation Door swings both ways

Very few dependencies between host OS and container apps

- Kernel API, esp. features like eBPF, iptables/nftables, cgroups v1/v2; containerd / docker
- As containerisation isolates applications from the OS, it also isolates the OS from applications
- Flatcar is only concerned with the OS
- Favourite app? Use a container. Tool missing? Container.

Connections to Kubernetes

- Node updates
- Kubernetes Installation options
- ClusterAPI

Service configurations

- Core services: systemd
- Everything else: control plane, out of scope

Cluster Operation

- Custom etcd reboot orchestration via Locksmith
- Everything else is out of scope

Wrap-up: Secure by Default

Security Considerations wrap-up

Secure by Default

Minimal attack surface, small footprint with very few run-time services

```
systemd[-resolved|-timesyncd|-udevd|-networkd|-userdbd|-logind], also for user sessions; dbus-daemon (system)
update_engine
containerd (and potentially docker)
sshd (if socket-activated)
locksmithd (if not on Kubernetes)
```

Secure workloads, secure update process

- Read-only, dm-verity protected OS partition
- Reproducible deployments w/ declarative configuration, no configuration drift
- User vs. shipped configuration tracking and management
- Updates are manually signed in air-gapped environment w/ HSM key

Secure Build and Release

- Sandboxed release builds on private servers, inputs are integrity checked
- SLSA-attested builds (deep dive in Appendix); signed SBOM 2.2 shipped with all OS releases
- Signed images and updates
- Access to release servers limited to few trusted maintainers, reviewed regularly

"Security First" Development Process

- Active CVE tracking by dedicated Security task force
- Security Primary / Secondary drive daily research / engagement

Thank You

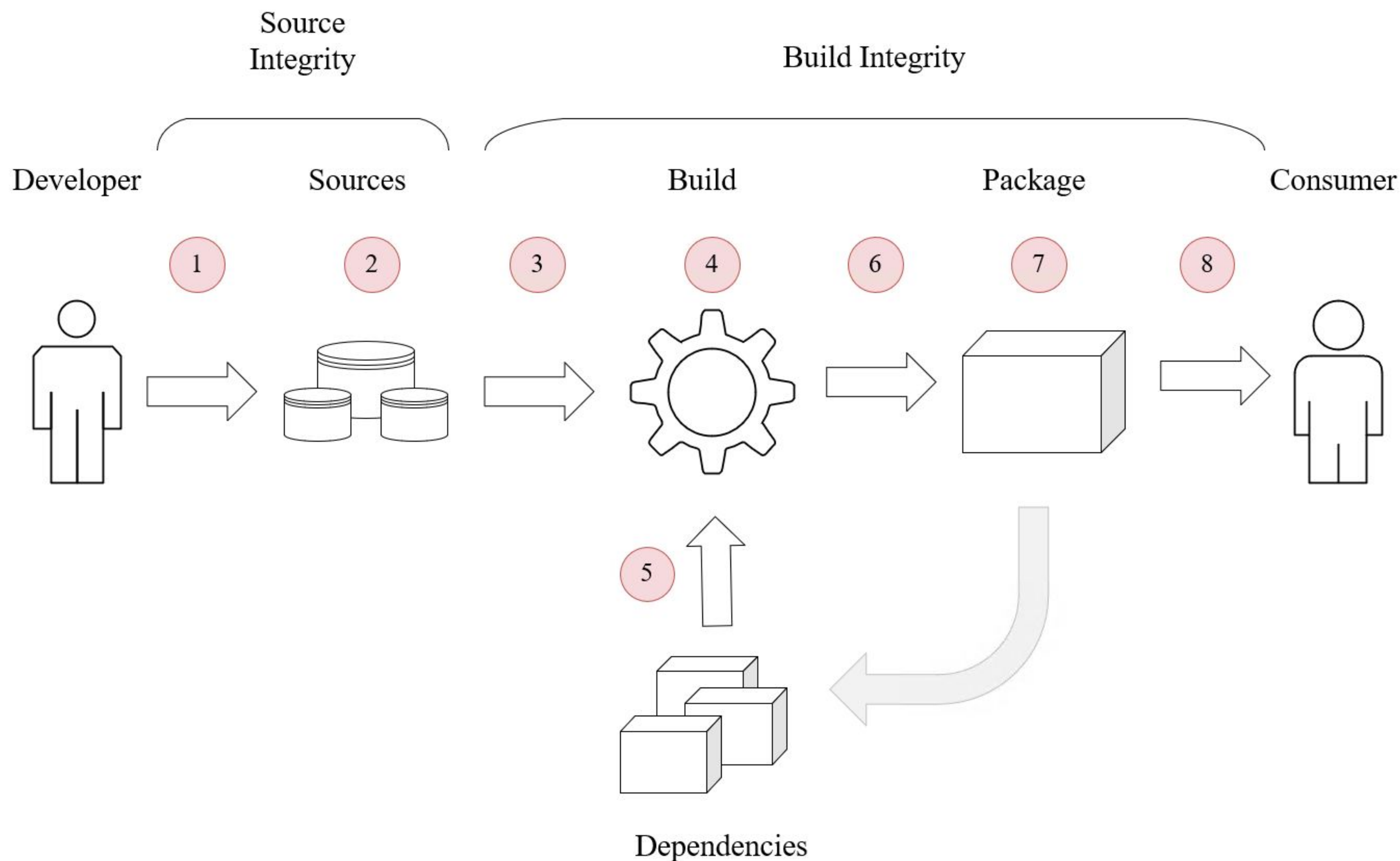
The Community's
Container Linux

Appendix

SLSA Deep Dive

Security Considerations

Supply Chain Security

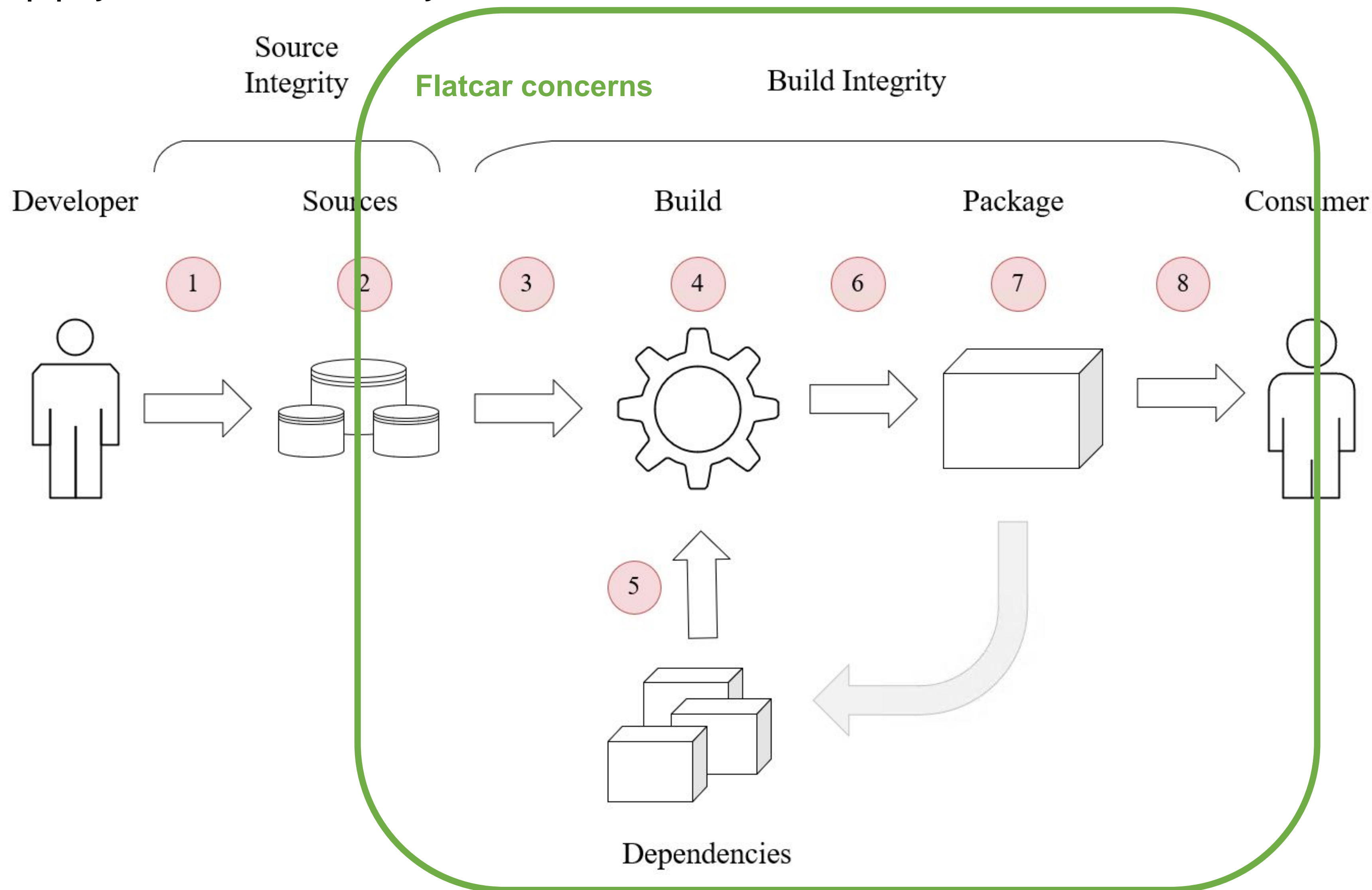


SLSA defines a number of key threats against supply chains:

1. unauthorised changes to sources
2. compromised source repositories
3. builds from a modified source
4. a compromised build process
5. use of a compromised dependency
6. publishing of a compromised package or image
7. a compromised package or image repository
8. injection / use of a compromised package or image

Security Considerations

Supply Chain Security

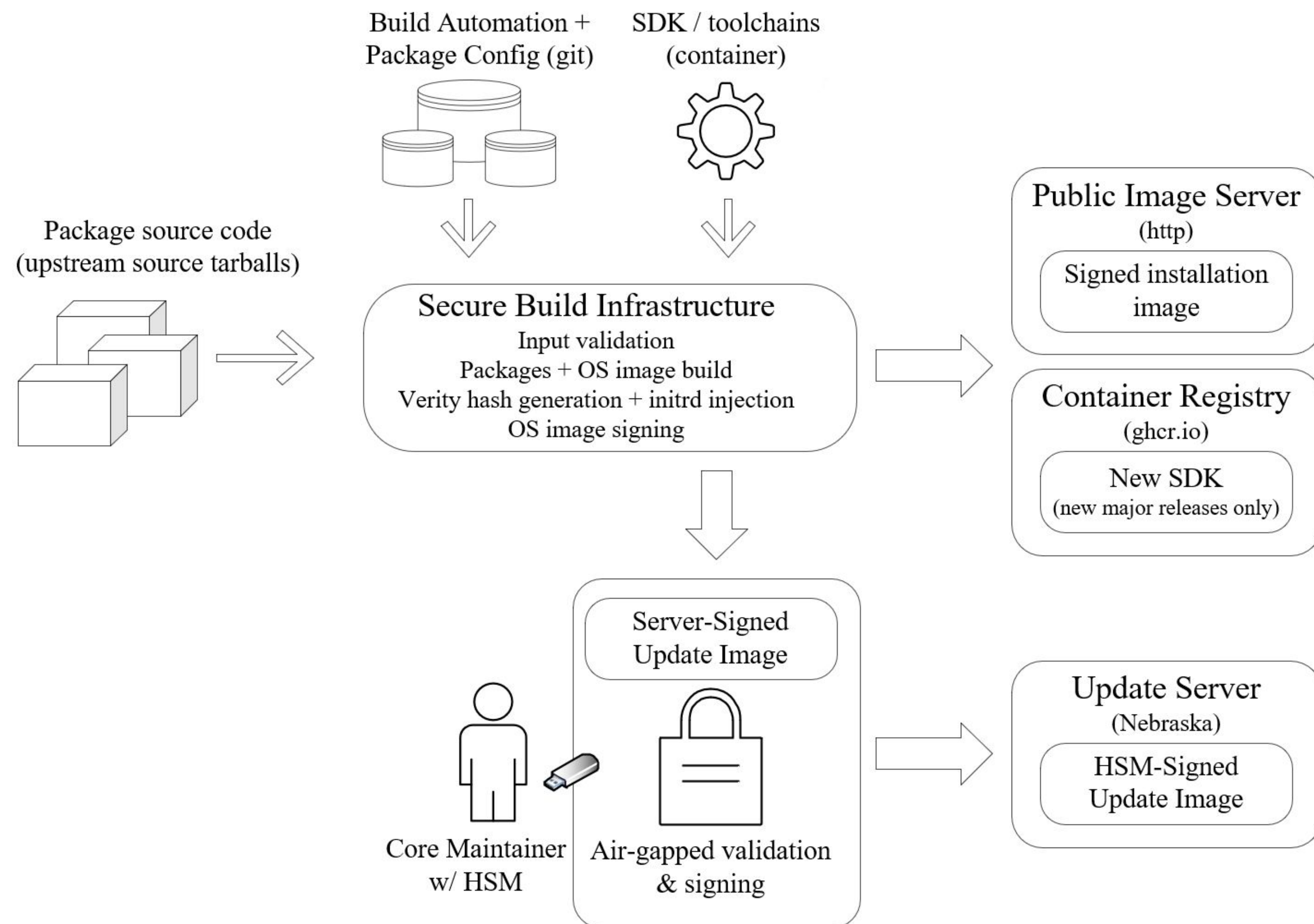


SLSA defines a number of key threats against supply chains:

1. unauthorised changes to sources
2. compromised source repositories
3. builds from a modified source
4. a compromised build process
5. use of a compromised dependency
6. publishing of a compromised package or image
7. a compromised package or image repository
8. injection / use of a compromised package or image

Security Considerations

Supply Chain Security – Build time



Inputs

1. Flatcar's build automation and package definition repository. Write access is limited to trusted group of core Flatcar maintainers. All changes are peer reviewed.
2. Upstream source tarballs of applications and libraries; Integrity is secured by cryptographic checksums.
3. The SDK container. Result of a previous build, validated by its container registry checksum.

Process

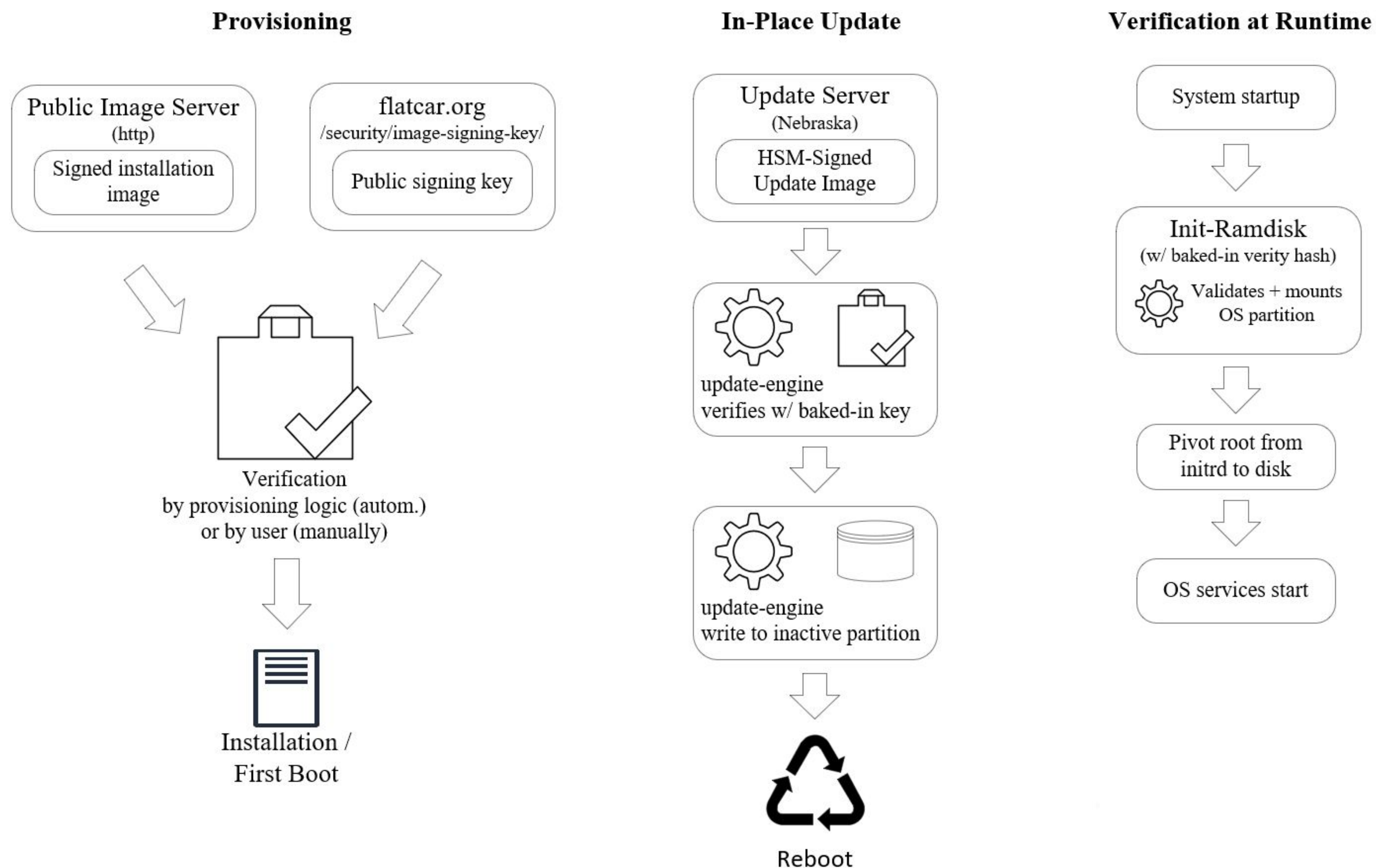
1. OS image and optionally SDK are built from validated sources on a dedicated bare metal machine in a secure, access-controlled Equinix Metal data center. Access to the infrastructure is limited to a small number of core maintainers and secured via a VPN.
2. Integrity of source tarballs is validated against multiple cryptographic checksums stored in input 1.
3. Per-package SLSA provenance is generated during build.
4. SLSA provenance is included in the signed OS image.
5. Additionally, a verity hash of the OS partition is generated and injected into the initrd for tamper protection at run time.
6. The update package is additionally signed manually by a trusted maintainer in an air-gapped environment. Only a small sub-set of core maintainers are authorized to perform this step.

Outputs

1. Signed OS images for all supported vendors.
2. A signed OS update package.
3. Optionally, a checksum-verified SDK container.

Security Considerations

Supply Chain Security – Provisioning and Runtime



Provisioning time

1. Images are signed with Flatcar's image signing key. Public key is available from our website.
2. Image integrity can thus be validated by provisioning logic before provisioning.

In-Place Upgrade

1. OS partition, Kernel, and initrd are combined in a single update package which was signed manually by a trusted maintainer in an air-gapped environment with a key stored on a HSM device.
2. Upgrade is downloaded by the Flatcar update client, verified against a compiled-in key in the client. The client resides on the read-only dm-verity protected OS partition.
3. New version is staged only after successful verification and activated via reboot.

Runtime

1. Kernel + Initrd (combined in a single binary blob) has baked-in verity hash for OS partition.
2. Initrd validates and mounts OS partition and pivots to the boot disk root.
3. Regular start-up commences. All OS services started reside on the read-only verity-protected OS partition.

Security Considerations

Supply Chain Security – Conformance

SLSA requirement	SLSA level 1	SLSA level 2	SLSA level 3	SLSA level 4	Flatcar meets
Source integrity: Source is version controlled		✓	✓	✓	✓
Source integrity: Source has verified history			✓	✓	✓
Source integrity: Source is retained indefinitely			18 months	✓	✓
Source integrity: Source is two-person reviewed				✓	✓
Build integrity: Scripted build	✓	✓	✓	✓	✓
Build integrity: Build service is used		✓	✓	✓	✓
Build integrity: Build as code			✓	✓	✓
Build integrity: Built in ephemeral environment			✓	✓	✓
Build integrity: Isolated			✓	✓	✓
Build integrity: Parameterless				✓	✓
Build integrity: Hermetic				✓	– [1]
Build integrity: Reproducible				Best Effort	✓ [2]
Provenance: Available	✓	✓	✓	✓	✓
Provenance: Authenticated		✓	✓	✓	✓
Provenance: Service generated		✓	✓	✓	✓
Provenance: Non-falsifiable			✓	✓	✓
Provenance: Dependencies complete				✓	✓
Common - Security				✓	– [3]
Common - Access				✓	✓
Common - Superusers				✓	– [4]

Notes

1. Build integrity - Hermetic builds: While Flatcar includes the potential for hermetic builds today - all sources are known in advance and can be staged to a build machine isolated from the network - the current build infrastructure and automation does not implement this feature. A tracking issue exists to address this in the future.
2. Build integrity - Reproducible: Many software packages such as compilers and core libraries insert build-variable information such as timestamps, user IDs, and host names into their binaries during the build process. While Flatcar's builds are 100% reproducible, the output may differ in a bit-by-bit comparison ONLY in places where this volatile information is compiled into the binaries.
3. Common - Security: This SLSA requirement is marked TBD in the SLSA standard and is not well defined at the time of writing; the essence appears to gravitate around a verifiable tamper-proof build infrastructure, e.g. via a full chain of trust. Flatcar is built on Flatcar to benefit from all the security features the distribution already ships with (discussed in detail below) - immutable OS binaries, boot time integrity check, etc. However, Flatcar currently does not support setting up a full chain of trust via TPM. A roadmap item aims to add TPM support to Flatcar, and have the build infrastructure support a full chain of trust.
4. Common - Superusers: The number of users with direct access to build infrastructure is very small, and users are well trusted. However, changes to the build system do not enforce approval by a second administrator.

Vendor Support

Appendix

Vendor Support

Flatcar supports

- EC2, Azure, GCE, Equinix Metal, and Digital Ocean clouds out of the box
  - Also available via AWS, GCP, and Azure Marketplaces
- VMWare, OpenStack, QEMU, LibVirt private clouds out of the box
- Community support further includes VirtualBox, Vagrant, Exoscale, Rackspace Cloud, Vultr, and Eucaliptos
- Hetzner is supported via Terraform (no Metadata support)

Vendor images

- Are derived from the a release's base OS build (i.e. no rebuild per vendor)
- Include vendor metadata support (bootstrap config) and vendor guest tools.
  - e.g. wa-agent for Azure image, open-vm-tools for VMWare, etc.
- Vendor tools are installed in a separate "OEM partition", base OS remains unchanged
- Vendor image generation (both fully supported and community) integrated in build process

Adding Vendor Support

- Vendor support is constantly extended; e.g. currently working on OVH
- Metadata, guest tools (where applicable); integration in Vendor clouds / marketplaces