Composer Deep Dive

Nils Adermann · @naderman
Private Packagist · packagist.com
Symfony Usergroup Berlin February 2026

— 0m depth

The Surface

The Big Picture

The Three Files

`composer.json`

What I Want

Your intent — required packages, version constraints, configuration

→

`composer.lock`

What Was Decided

Flattened dependency graph

- transitive dependencies
- exact versions
- checksums (if available)
- download URLs

→

`vendor/`

What's Installed

The actual code on disk — autoloader, package files

(yes it’s a directory 😛)

composer update & install

`composer update`

1. Reads composer.json + repositories
2. Runs the dependency solver
3. Writes composer.lock
4. Runs a composer install

`composer install`

1. Reads composer.lock
   ⚠️ Without lock file → runs an update!
2. Downloads & installs to vendor/
3. No solving — exact versions only

`composer require / remove`

`require` = edit composer.json (add package) + run update

`remove` = edit composer.json (remove package) + run update

Separating update & install

A common scenario: vendor, lock and json are at different versions

vendor/

v8.0.5

previous local attempt

composer.lock

v7.3.11

current production state

composer.json

^7.4

intended upgrade constraint

```
$ composer update
Lock file operations: 0 installs, 1 update, 0 removals
  - Upgrading symfony/http-foundation (v7.3.11 => v7.4.5)
Writing lock file

Package operations: 1 update, 1 removal
  - Downgrading symfony/http-foundation (v8.0.5 => v7.4.5)
```

update writes the lock based on json constraints, then install syncs vendor — including downgrades!

Commit the Lock File!

If you don't commit composer.lock ...

- composer install without a lock file runs an update
- You're not managing your dependencies — they do whatever they want
- Conflicts can randomly occur on install
- You may not get the same code on different machines

The lock file exists to be committed!

— 20m depth

Shallow Waters

Commands & Options You Should Know

composer bump

Since Composer 2.4 — raises the lower bound to the currently installed version

```
// Before bump
"require": { "laravel/framework": "^11.0" }

$ composer bump

// After bump
"require": { "laravel/framework": "^11.38.2" }
```

Benefits

- Prevents accidental downgrades
- Slightly improves resolver performance
- `Supports --dev-only` for libraries

Not recommended for libraries — narrows constraints for downstream users

composer why / why-not

`composer why` (depends)

"Which packages depend on X?"

```
$ composer why symfony/console
laravel/framework  v11.0  requires
  symfony/console (^7.0)
```

`composer why-not` (prohibits)

"Why can't I install version Y?"

```
$ composer why-not symfony/console 8.0
laravel/framework  v11.0  requires
  symfony/console (^7.0)
```

Essential for debugging upgrade blockers

"I want to upgrade package X to version 2.0 but composer won't let me — which of my other dependencies is blocking it?"

composer audit

Since Composer 2.4 — vulnerability scanning built in

- Lists vulnerable versions in composer.lock
- Uses packagist.org vulnerability DB API
  - → GitHub advisory database
  - → FriendsOfPHP/security-advisories
- Non-zero exit code → use in CI pipelines
- composer update runs audit by default

```
$ composer audit

Found 2 security vulnerability
  advisories affecting 2 packages:

Package: symfony/http-kernel
  CVE-2024-50340
  Severity: high
```

Composer 2.9: Automatic Security Blocking

Composer now blocks vulnerable packages during dependency resolution by default

How it works

- Vulnerable versions are discarded before resolving dependencies
- Prevents installing packages with known security advisories
- Enabled by default — no action needed
- Can temporarily ignore specific advisories
- Optional: also block abandoned packages via audit.block-abandoned

Configuration

```
"audit": {
  "block-insecure": true,
  "block-abandoned": false
}
```

Disable per-run:
`COMPOSER_NO_SECURITY_BLOCKING=1`
or `--no-security-blocking`

If you were using roave/security-advisories, this feature replaces it — you can remove that dependency

Partial Updates & Dependencies

```
composer update vendor/pkg
```

Update just this one package

```
composer update vendor/pkg -w
```

Also update its dependencies, but not direct dependencies of the project in your composer.json

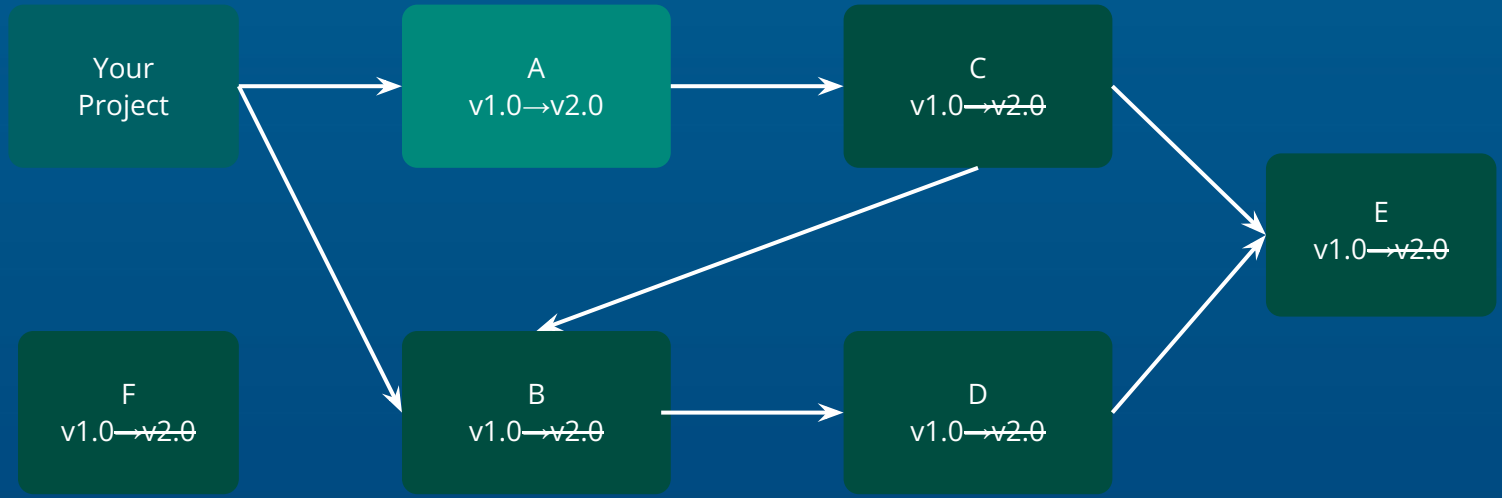
```
composer update vendor/pkg -W
```

Update ALL transitive dependencies, including ones listed in composer.json

-w = --with-dependencies **-W** = --with-all-dependencies

Partial Updates

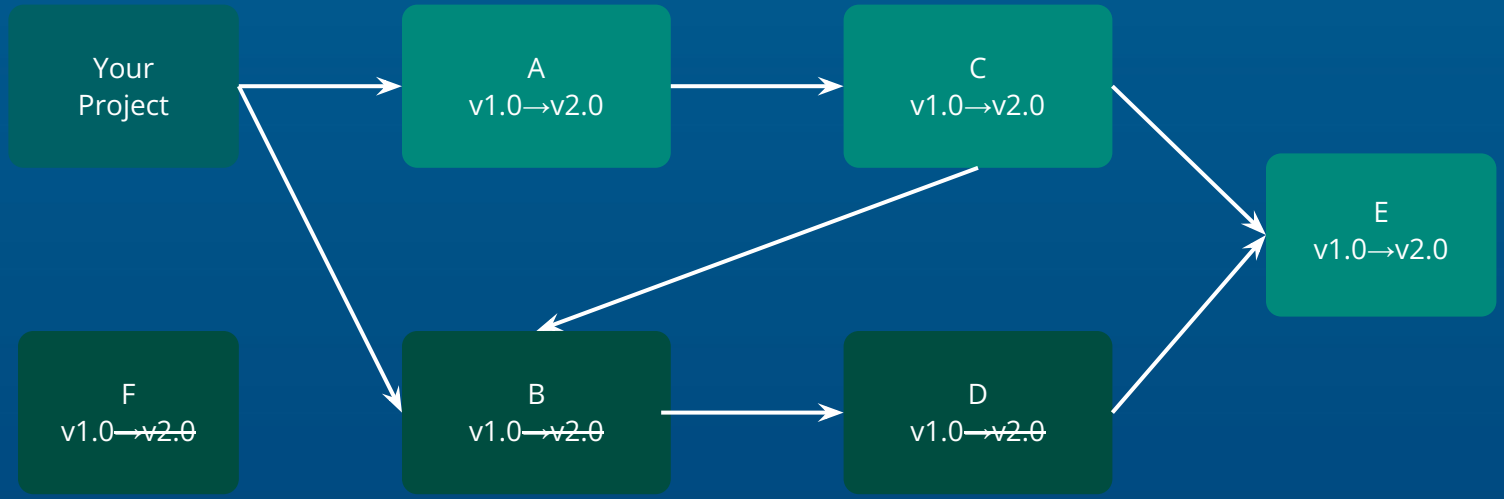
Dependency graph for a partial update of package A



`composer update A` → only updates A. If A v2.0 requires C v2.0, it fails.

Partial Updates

Dependency graph for a partial update of package A

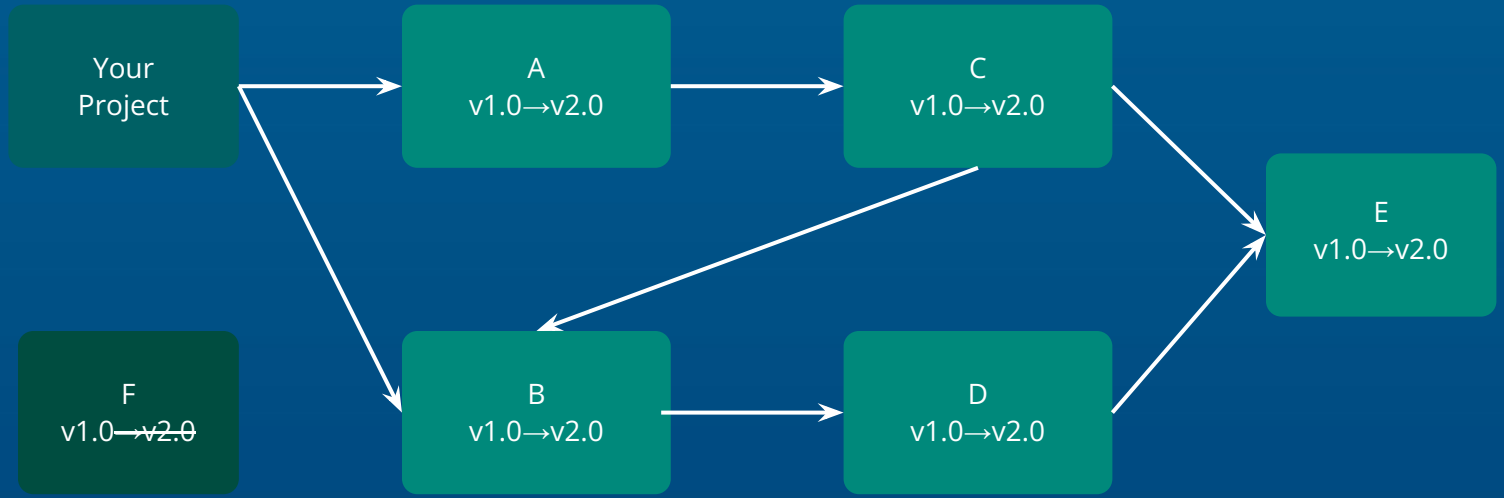


`composer update A` → only updates A. If A v2.0 requires C v2.0, it fails.

`composer update A -w` → also updates C and E (transitive deps of A). B stays at 1.0 and D stays at v1.0.

Partial Updates

Dependency graph for a partial update of package A



`composer update A` → only updates A. If A v2.0 requires C v2.0, it fails.

`composer update A -w` → also updates C and E (transitive deps of A). B stays at 1.0 and D stays at v1.0.

`composer update A -W` → updates A, B, C, D and E

--minimal-changes

Since Composer 2.7 (Feb 2024)

The Problem

"I want to update one dependency, but there's a conflict. I need to update more deps, but I don't want to update everything."

The Solution

Partial updates with dependencies, but keeping packages at the lock file version if possible

How it works under the hood

1. Set up the update the same way as without the option
2. Make the policy pick locked version numbers before any other
3. Solver tries locked versions first → only changes if incompatible

--minimal-changes — Example

You want to update A. A v2.0 requires C ≥3.0, but C is locked at v2.5.

`composer update A -W`

A v1.0 → v2.0
C v2.5 → v3.2 (latest)
D v1.0 → v1.5 (latest)
E v4.0 → v4.3 (latest)

Updates everything to latest — risky!

`composer update A -W -m`

A v1.0 → v2.0
C v2.5 → v3.0 (needed to update to A v2.0)
D v1.0 (unchanged — locked)
E v4.0 (unchanged — locked)

Only changes what's absolutely necessary!

```
composer update vendor/pkg -W --minimal-changes
// or short: composer update vendor/pkg -W -m
```

The Platform Repository

An implicit, read-only repository

Contains "packages" representing your system:

- PHP version (php, php-64bit, php-ipv6)
- PHP extensions (ext-json, ext-curl, ...)
- System libraries (lib-openssl, lib-curl, ...)

These cannot be updated/installed/removed

They reflect the actual runtime environment

```
$ composer show --platform
php           8.3.12
php-64bit     8.3.12
ext-curl      8.3.12
ext-json      8.3.12
lib-openssl   3.0.13
...
```

config.platform — simulate a different environment

```
"config": { "platform": { "php": "8.1.6", "ext-intl": "1.1.0" } }
```

→ Composer solves as if you're on PHP 8.1.6, even if your dev machine runs 8.3

--ignore-platform-req and the + suffix

`--ignore-platform-reqs`	Ignores ALL platform checks *Dangerous — may install packages missing required extensions*
`--ignore-platform-req=php`	Ignores only PHP version *Still checks extensions — but ignores both lower AND upper bounds*
`--ignore-platform-req=php+`	Ignores only the PHP upper bound *The correct flag for testing on newer PHP versions*

Example: package requires `php: ^8.3` (matches ≥8.3.0 and <9.0.0)

Testing on PHP 9.0: **`php+`** ignores the <9.0 upper bound → ✓ installs fine

Testing on PHP 8.1: **`php+`** still enforces the ≥8.1 lower bound → ✗ fails correctly

More Commands Worth Knowing

`composer outdated`

Lists outdated packages — try --major-only, --minor-only

`composer diagnose`

Checks for common problems with your setup

`composer fund`

Shows funding links for your dependencies

`composer reinstall vendor/*`

Clean reinstall — useful if you modified vendor files

`composer check-platform-reqs`

Verify platform requirements — run this on deploy!

— 50m depth

Mid-Depth

Underlying Concepts & Security

Composer Plugins & Security

`config.allow-plugins`

Since Composer 2.2

Must explicitly enable each plugin

Protects from executing malicious code before you review lock file changes

Root-Only for Scripts & Plugins

Limited to root composer.json only

→ Protects from malicious maintainers

→ Prevents dependency confusion attacks

→ Blocks accidental dependency execution

You still have to review your lock file changes!

Why Dependencies Cannot Define Repositories

Repository definitions are only read from the root composer.json

Security

If dependencies could add repos, an attack one package's composer.json quickly widens scope: They could overwrite other packages

Solvability

With "provide" and "replace", the full package graph is unknown upfront. Allowing deps to add repos would either lead to paradox or load repos from old versions with no way to remove

Determinism

The project owner controls where packages come from — all sources visible in one place

Lock File Hash & Resolving Conflicts

content-hash

Computed over relevant parts of composer.json

Ensures install warns when json & lock are out of sync

The lock file WILL conflict in git merges — by design

How to resolve lock conflicts

1. `git checkout main -- composer.lock`
   (take one side)
2. `composer update <changed deps>`
   (re-apply your changes)
3. `Put exact command in commit message`

```
{
  "_readme": ["This file locks the dependencies..."],
  "content-hash": "a1b2c3d4e5f6789...",
  "packages": [ ... ]
}
```

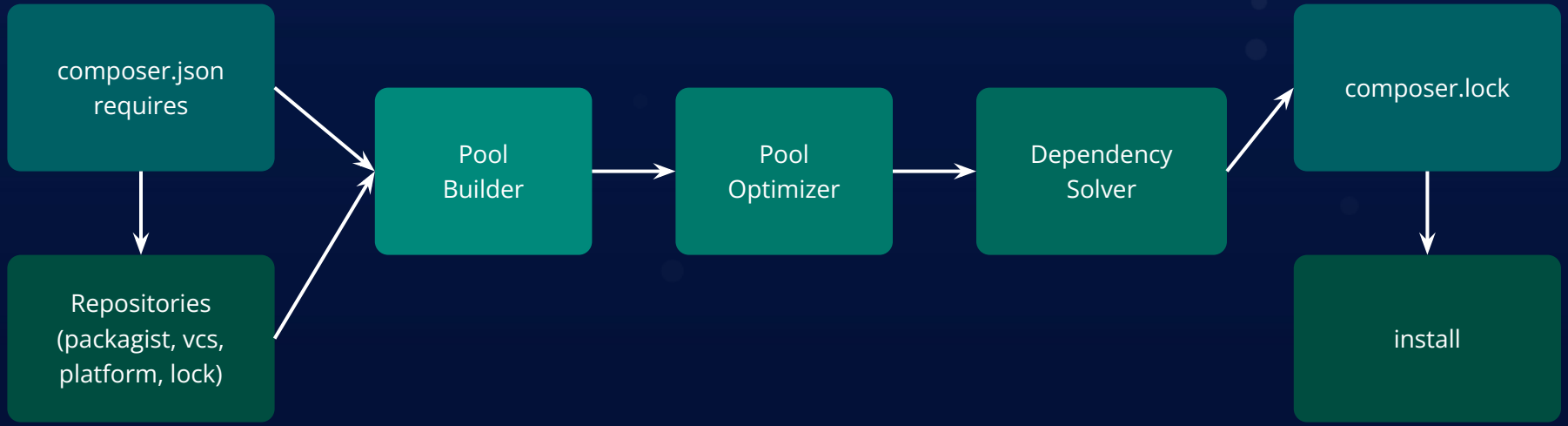
Never manually edit composer.lock — always regenerate it with composer update

— 200m depth

The Deep

Dependency Resolution & the SAT Solver

composer update — The Full Pipeline



Pool Builder collects metadata, avoids loading definitely-uninstallable packages

Pool Optimizer merges versions with identical constraints, filters impossible packages

Dependency Solver uses SAT solving to find a valid set of packages to install

What's in the Dependency Solver?

SAT Solver

Boolean SATisfiability Problem

"Is there a set of values for a boolean formula that makes it evaluate to true?"

(A ∧ B) → satisfiable (A=T, B=T)

(A ∧ B ∧ ¬A) → not satisfiable

Why a SAT Solver?

- Ported from libzypp / zypper (SUSE) in 2011
- EDOS project proved:
  Package installation is NP-Complete
- Any 3SAT problem can be encoded as a package installation problem

  `https://www.mancoosi.org/edos/`

Dependencies as a SAT Problem

Project requires A — A requires B and C — B conflicts with C

1. (A-1.0.0) ∧ (¬A-1.0.0 ∨ B-1.0.0) ∧ (¬B-1.0.0 ∨ ¬C-1.0.0) ∧ (¬A-1.0.0 ∨ C-1.0.0)

2. A-1.0.0 = true
   true ∧ (B-1.0.0) ∧ (¬B-1.0.0 ∨ ¬C-1.0.0) ∧ (C-1.0.0)

3. B-1.0.0 = true
   true ∧ true ∧ (¬C-1.0.0) ∧ (C-1.0.0)

4. C-1.0.0 = false
   true ∧ true ∧ true ∧ false

   ⚡ Conflict! A requires C, but B conflicts with C.

The solver detects this conflict and backtracks to try different versions

Composer 2: Multi-Conflict Rules

"Only one version of a package can be installed" → implies conflicts between every pair

Versions	Composer 1	Composer 2
3	3 rules	1 rule
6	15 rules	1 rule
100	4,950 rules	1 rule
500	124,750 rules	1 rule
1,000	499,500 rules	1 rule

Composer 2 uses a single `oneof()` multi-conflict rule — this is a key reason Composer 2 is so much faster

The Solver Implementation

Key Classes

`Rule`
Array of literals — abs value = package id, sign = negation

`Pool`

Simple array of all package versions

`Solver::solve()`
Generates rules from pool + policy → finds solution with runSat()

`DefaultPolicy`
Implements free choice decisions — handles --prefer-lowest, --prefer-stable

--minimal-changes revisited

Now you can understand how it works:

1. Set up the update normally
2. Modify the DefaultPolicy to pick locked version numbers before any others
3. Solver tries locked versions first
4. Only changes if locked versions are incompatible

```
github.com/composer/composer/pull/11665
```

— ascending...

Part 5: Resurfacing

Outlook & Future Plans

Upcoming Features & Future Plans

Stability & Compatibility

Keep things stable — no unnecessary breaking changes

Workflow Improvements

Small improvements based on common developer workflows

Security: Malware Feed

Plans to integrate filtering based on malware feed from Aikido

OIDC Authentication

Plans to support OIDC auth for private Composer repositories like Private Packagist

Artifact Integrity

Plans to host artifacts on packagist.org with sigstore and trusted publishing to ensure build integrity

Thank You!

Nils Adermann · @naderman

Private Packagist — packagist.com

Composer — getcomposer.org