



IP PARIS



Supply chain attacks and Rust

NET7212 — Safe System Programming (in Rust)

Samuel Tardieu Stefano Zacchiroli

2023-12-05




Open Source Software Security

Open Source is everywhere

- Recent analyses¹ by major industry players in the field of mergers and acquisitions (M&A) software audits report that **99% of code bases** audited in 2019 **contained open source software** components, with **70%** of all audited code **being itself open source**.
- More open source software around → more vulnerabilities found in it.
- All other factors being equal—e.g., security practices, programming technologies, funding, people power, etc.—**open source software is *more secure*** than proprietary software, not less.²

¹Synopsis: [2020 open source security and risk analysis report \(OSSRA\)](#) . Tech. rep., Synopsis (2020). With minor variations, these numbers have been confirmed year after year for almost a decade now.

²For a list of pro/con arguments in this long-lasting debate see: Christian Payne: [On the security of open source software](#) . Inf. Syst. J. 12(1): 61-78 (2002).

Open Source Software Supply Chain Attacks

References

- [Ohm20]: Marc Ohm, Henrik Plate, Arnold Sykosch, Michael Meier. *Backstabber's Knife Collection: A Review of Open Source Software Supply Chain Attacks* [↗](#). DIMVA 2020: 23-43.
- [Ladisa22]: Piergiorgio Ladisa, Henrik Plate, Matias Martinez, Olivier Barais: *SoK: Taxonomy of Attacks on Open-Source Software Supply Chains* [↗](#). IEEE S&P 2023: 167-184.

The software supply chain

- **Supply chain:** the set of activities required by an organization to deliver goods or services to consumers.
- **Software supply chain:** the set of *software components and software services* required to deliver an IT product or service to users.
 - Libraries, runtimes, and other software component *dependencies*
 - *Base system* (operating system, package manager, compiler, ...)
 - *Development* tools and platform (e.g., IDEs, build system, GitHub/GitLab, CI/CD, ...)
 - etc.
- Key artifact for audits: SBOM = Software Bill of Materials
 - “A SBOM is a nested inventory, a list of ingredients that make up software components.”³

³<https://www.cisa.gov/sbom>

Supply chain attacks

A **software supply chain attack** is a particular kind of *cyber-attack* that aims at *injecting malicious code* into an otherwise *legitimate software product*.

Notable examples

- *NotPetya* (2017): ransomware concealed in an update of a popular accounting software, hitting Ukrainian banks and major corps (B\$).
- *CCleaner* (2017): malicious version of a popular MS Windows maintenance tool, distributed via the vendor website.
- *SolarWinds* (2020): malicious update of the SolarWinds Orion monitoring software, shipping a delayed-activation trojan. Breached into several US Gov. branches as well as Microsoft.



Open source supply chain attacks

- Is this specific to Free/Open Source Software (FOSS)? No.
- But modern **FOSS package ecosystems** are heavily intertwined.
 - Examples: NPM (JavaScript), PyPI (Python), Crates (Rust), Gems (Ruby), etc.
 - 100/10k/1M packages, depending on each other due to code reuse opportunities.
 - **Reverse transitive dependencies** grow fast. A single package could be required by *thousands* of others.

left-pad (2016)

(Not an attack, but gives an idea of how entangled package ecosystems could be.)

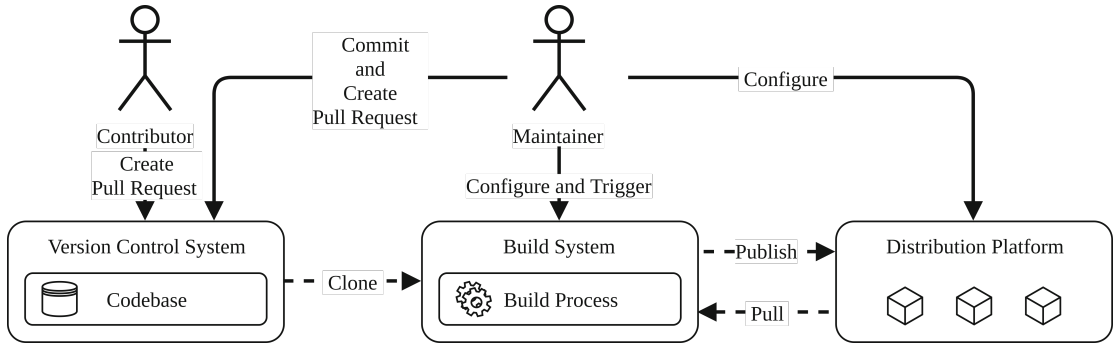
```
function leftpad (str, len, ch) {
  str = String(str);
  var i = -1;
  if (!ch && ch !== 0) ch = ' ';
  len = len - str.length;
  while (++i < len) { str = ch + str; }
  return str;
}
```

- Maintainer: *“I have the right to delete my stuff”*. “Unpublish” package.
- Impact: “many thousands of projects”—including major ones like babel and atom—no longer installable with `npm`.
- NPM repository operators (a for-profit company) forcibly “un-unpublish” package.

Open source supply chain attacks (cont.)

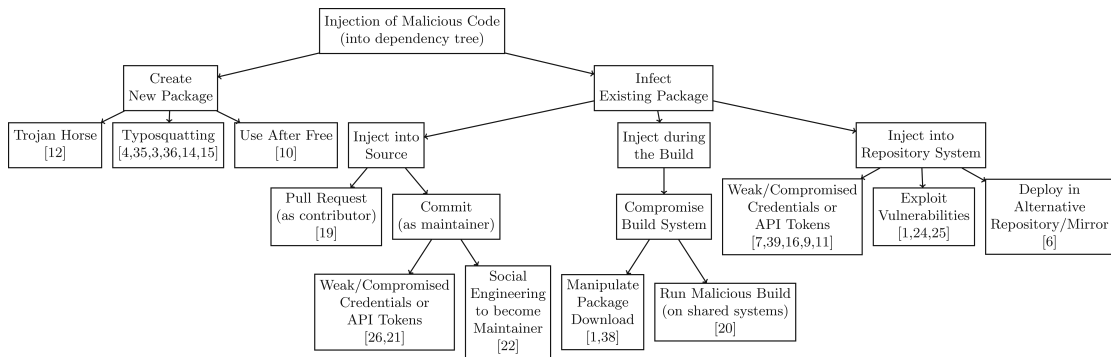
- For an attacker, code injection into (transitively) popular leaf packages has a **low opportunity cost**.
- Also, entirely open FOSS package ecosystems (\neq Linux distros) can be **easy to infiltrate**.

(An) open source development workflow



(image from [Ohm20](#))

Attack tree — Injection



(image from [Ohm20])

Attacker's goal: package P containing malicious code is available from download from a distribution platform *and* P is a reverse transitive dependency of a legitimate package.

Attack vector — Typosquatting

Injection → Create New Package → Typosquatting

1. Create a **new package** with a **name similar** (e.g., Levenshtein distance ≤ 2) to an existing popular package, including malicious code. Examples:
 - Squat on PyPI the Debian package name (“python-sqlite” v. “sqlite”)
 - English variants (“color” v. “colour”)
 - Unicode tricks
2. Upload it to a distribution platform (e.g., PyPI).
3. Wait for users to mistype (e.g., `pip install python-sqlite`).

Related attack vector: **Use After Free**

Attack vector — Become maintainer

Injection → Infect Existing Package → Inject into Source → Commit (as maintainer) → Social Engineering to become Maintainer

1. Package maintainer: “I no longer have time for this project, who wants to take over its maintenance?”
2. Attacker: raises hand.
3. Attacker: releases new version including malicious code.

Might require early investment by the attacker to accrue enough “street credibility” to win over maintenance at the right moment. For popular packages with low bus factor it could be worth it.



Attack vector — Compromise build system

Injection of Malicious Code → Infect Existing Package → Inject during the Build → Compromise Build System

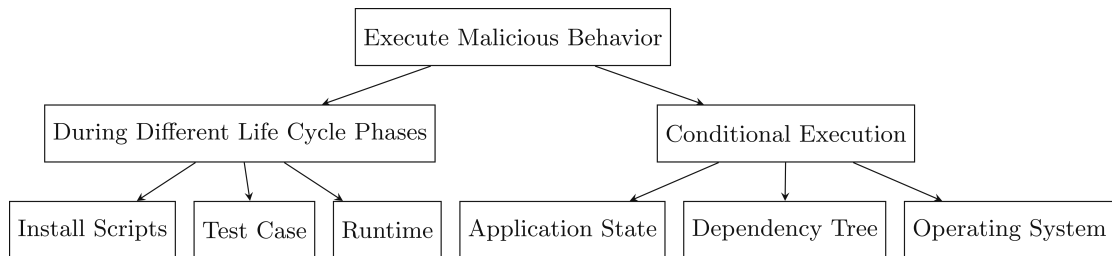
- Often, code run by users is *written but not built* by maintainers.
- Rather, it is built by **3rd-party vendors**.
 - E.g., GNU/Linux distros, app store operators, arch “porters”.
- It hence becomes attractive to **break into vendor build systems**, compromising binaries “downstream”, without anybody auditing source code noticing.

Attack vector — Exploit vulnerabilities

Injection of Malicious Code → Infect Existing Package → Inject into [Package] Repository System → Exploit Vulnerabilities

- Attack on the package repositories (\neq VCS repositories) used to distribute packages to the final users.
- Use known vulnerabilities to break into the package repository host.
- Modify packages (without modifying their metadata) injecting malicious code that will trigger on target systems.
- Examples:
 - [Remote Code Execution on rubygems.org](#)  (2017)
 - [Remote Code Execution on packagist.org](#)  (2018)

Attack tree — Execution

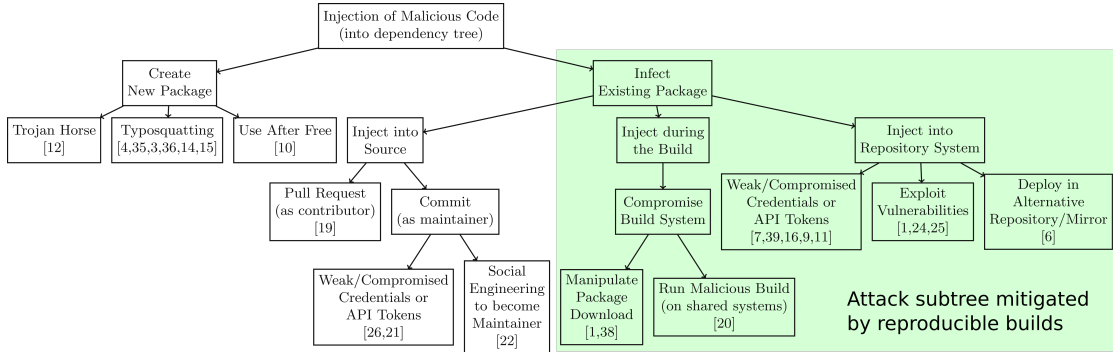


(image from [Ohm20])

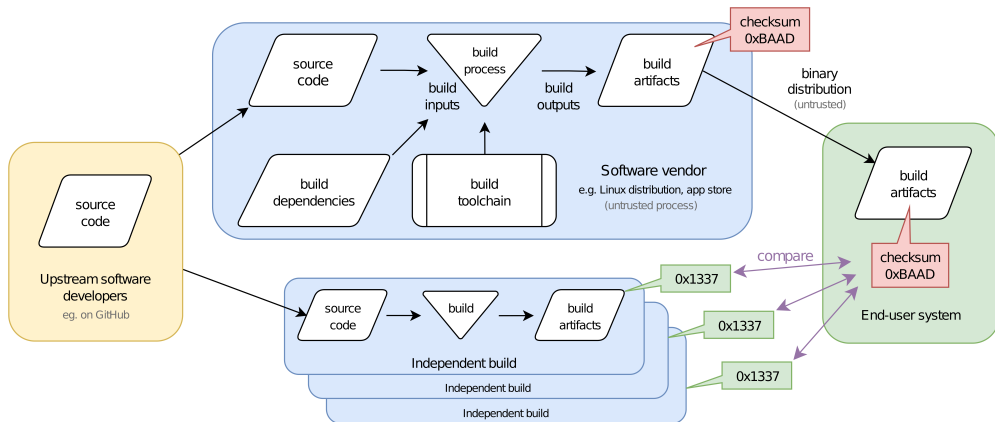
- There is a lot of variability in **when the attack is executed**, in terms of software life cycle (install/ test/ runtime).
- There is also variability in **if the attack is executed**, e.g., to target specific users and lower the chances of being detected in other contexts.

Trusting 3rd-party builds

How can we increase users' trust when running (trusted) FOSS code built by (untrusted) 3rd-party vendors?



Reproducible builds



Learn more: [Reproducible Builds project](#) and paper.⁴

⁴Chris Lamb, Stefano Zacchiroli: [Reproducible Builds: Increasing the Integrity of Software Supply Chains](#). IEEE Softw. 39(2): 62-70 (2022).

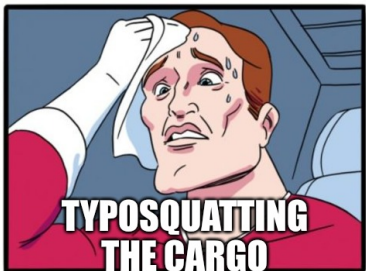
Rust and supply chain attacks

How do Rust and its ecosystem compare to other languages/ecosystems in terms of supply chain security?

- Poll: who think Rust is doing better/worse/same than others?
- Do Rust safety features help against this class of attacks?
- What's the root cause of all these problems?

Typosquatting in the Rust ecosystem — example

Security advisory: malicious crate rustdecimal [↗](#)



The Rust Security Response WG and the crates.io team were notified on 2022-05-02 of the existence of the malicious crate rustdecimal, which contained malware. The crate name was intentionally similar to the name of the popular rust_decimal crate, hoping that potential victims would misspell its name (an attack called “typosquatting”).

To protect the security of the ecosystem, the crates.io team permanently removed the crate from the registry as soon as it was made aware of the malware. [...]

The crate had less than 500 downloads since its first release on 2022-03-25, and no crates on the crates.io registry depended on it.

*The crate contained identical source code and functionality as the legit rust_decimal crate, except for the Decimal::*new* function.*

When the function was called, it checked whether the GITLAB_CI environment variable was set, and if so it downloaded a binary payload into /tmp/git-updater.bin and executed it. The binary payload supported both Linux and macOS, but not Windows.

An analysis of the binary payload was not possible, as the download URL didn't work anymore when the analysis was performed.

Root cause

- Low to absent **gatekeeping** in public package manager repositories
 - Everyone can create an account and upload (no *maintainer gatekeeping*)
 - Everyone can “claim” a package name → no polishing/audit of the **global package namespace** (no *namespace gatekeeping*)
- Compare and contrast with the case of traditional GNU/Linux distributions.
The case of *Debian*:
 - Package policies that covers package naming rules (see, e.g., the [Debian Python Policy](#))
 - New packages (and packages that modify the package namespace) go through manual review ([Debian NEW queue](#))
 - Strictly controlled set of Debian developers, with a formal “hiring” process (see the [Debian New Members process](#))
- Argument against gatekeeping: *go fast, reduce friction.*

You can't have it both way!


It's either more friction and more control. Or less friction and less control.

(Up to now, the software industry has favored reducing friction.)

Mitigations


- The state of the art in addressing open source supply chain attacks is based on various **mitigation techniques**. Key notions:
 1. **Know Your Software** (KYSW): have a full, clear understanding of all your software dependencies.
 - Ideally documented in a **SBOM** (Software Bill of Material).⁵
 2. **Audit** regularly your dependencies and cross-reference them with **knowledge bases** of quality information about them. Quality criteria: security, maintenance, best practices,⁶ licensing, etc.
- Note that KYSW could already be very hard!, due to the dependency entanglement we have seen.
- **Automation** is needed to minimize the risks of overlooking issues in the future.
 - Ideal workflow: at each CI/CD build you automatically list all your dependencies, compare them against up-to-date knowledge bases, raise warnings/failures if a known security vulnerability affects the built artifact.
 - Q: is this enough?

It is a complex technology space, which is moving very fast. In the following we will just highlight some of the existing tools in the Rust ecosystem that help in taming the software supply chain.

⁵[Software Bill of Materials](#) , NTIA, US Dept. of Commerce.

⁶Sample badging schemes: [OpenSSF best practices](#) , [OW2 best practices](#) .

Know your Rust stack — cargo tree

- We have seen how Cargo is both a *package manager* and a *build system*.
- One of the advantages of this joint design is that Cargo has a very clear view of all the (Rust) dependencies of a project.
- `cargo tree`  is a builtin Cargo command to show the **dependency tree** of a project.

Worth noting:

- Dependency deduplication (*)
- Detection of dependencies occurring in multiple versions (`--duplicates`)
- Q: Is this complete? Which dependencies could you be missing?

```
$ cd /myproject
$ cargo tree
myproject v0.1.0 (/myproject)
|-- rand v0.7.3
    |-- getrandom v0.1.14
    |   |-- cfg-if v0.1.10
    |   |-- libc v0.2.68
    |-- libc v0.2.68 (*)
    |-- rand_chacha v0.2.2
    |   |-- ppv-lite86 v0.2.6
    |   |-- rand_core v0.5.1
    |       |-- getrandom v0.1.14 (*)
    |-- rand_core v0.5.1 (*)
[build-dependencies]
|-- cc v1.0.50
```

The RustSec Advisory Database

- The **RustSec Advisory Database**⁷ is the official, public database of known vulnerabilities affecting the Rust ecosystem.
- Data model: a mapping from *crate names* and *versions ranges* to security advisories describing *known vulnerabilities*.
 - Variant: also supports documenting security issues affecting the Rust toolchain (e.g., the compiler).
 - Variant: also supports documenting affected functions within a crate, for finer-grained detection.
- Examples:
 - [RUSTSEC-2023-0021](https://rustsec.org/advisories/RUSTSEC-2023-0021)⁷: NULL pointer dereference in `stb_image` (C bindings)
 - [RUSTSEC-2023-0018](https://rustsec.org/advisories/RUSTSEC-2023-0018)⁷: Race Condition Enabling Link Following and Time-of-check Time-of-use (TOCTOU)
 - [RUSTSEC-2023-0015](https://rustsec.org/advisories/RUSTSEC-2023-0015)⁷: Ascii allows out-of-bounds array indexing in safe code (dangerous in `--release` mode)
 - [RUSTSEC-2021-0151](https://rustsec.org/advisories/RUSTSEC-2021-0151)⁷: `ncollide2d` is unmaintained

⁷Web site: <https://rustsec.org/>, raw data: <https://github.com/RustSec/advisory-db>

Your supply chain vs the advisory database — cargo audit

- `cargo audit` [🔗](#) is a cargo plugin command, maintained by the RustSec team (the same team that maintains the advisory database).
- Install: `cargo install cargo-audit`.
- Basic idea: compare your full dependency tree (a-la `cargo tree`) against the RustSec advisory database, emitting warnings/errors for each vulnerability (potentially) affecting your code base.
 - The full advisory DB is retrieved/updated at each audit.
 - Can be easily integrated into CI/CD toolchains.
 - Rely on a static view of your dependency tree (does not build/inspect your code).
 - Prone to both false positives and false negatives (why?).
- `cargo audit fix`: attempts to automatically “fix” your dependencies, by upgrading them to the next compatible (according to semantic versioning) and safe version. Note that it is not always possible to do so.

cargo audit — example

```
$ cargo init auditme
$ cd auditme/
$ cargo add stb_image@=0.2.4
$ cargo add lington
$ tail -n 3 Cargo.toml
[dependencies]
linton = "0.1.0"
stb_image = "=0.2.4"
$ cargo tree
[ huge dependency tree... ]
```

Demo

Note how `cargo audit` is not only about *security* vulnerabilities; it also warns about **maintenance** aspects, because they can turn into *future* vulnerabilities!

```
$ cargo audit
Fetching advisory database from `https://github.com/RustSec/advisory-database`
  Loaded 527 security advisories [...]
Scanning Cargo.lock for vulnerabilities (71 crate dependencies)

Crate:      stb_image
Version:    0.2.4
Title:      NULL pointer derefernce in `stb_image`
ID:         RUSTSEC-2023-0021
URL:        https://rustsec.org/advisories/RUSTSEC-2023-0021
Solution:   Upgrade to >=0.2.5

Crate:      xml-rs
Version:    0.8.4
Warning:    unmaintained
Title:      xml-rs is Unmaintained
ID:         RUSTSEC-2022-0048
URL:        https://rustsec.org/advisories/RUSTSEC-2022-0048

error: 1 vulnerability found!
warning: 1 allowed warning found
```

Auditing binaries — cargo audit bin and auditable

- In some cases you only have an executable **binary to audit** and would like to know if it is affected by a known vulnerability or not. `cargo audit bin` can do that. However:
- In general, executables do not carry with them full SBOM information.
 - In this case `cargo audit bin` will *heuristically* try to determine the packages and versions used by a Rust binary based on `panic!` message strings.
- A standard convention is under development⁸ to ship package version information within executable binaries.
 - It stores in a binary section (e.g., ELF on UNIX) a compressed JSON file including build-time dependency information; it takes just a few KiB for very large dependency trees.
 - `cargo auditable`⁹ is a Cargo build wrapper that store this information in built binaries.
 - `cargo audit bin` uses the information, if available.

⁸<https://github.com/rust-lang/rfcs/pull/2801>

⁹<https://github.com/rust-secure-code/cargo-auditable>

cargo audit bin and auditable — example

```
$ cargo build --release
$ cargo audit bin target/release/auditme
[...]
warning: target/release/auditme was not built with 'cargo auditable',
the report will be incomplete (8 dependencies recovered)
```

VS

```
$ cargo auditable build --release
$ cargo audit bin target/release/auditme
[...]
Crate:      stb_image
Title:      NULL pointer derefernce in `stb_image`
ID:         RUSTSEC-2023-0021
[...]
Crate:      xml-rs
Warning:    unmaintained
ID:         RUSTSEC-2022-0048
```

cargo deny

- Auditing your software supply chain is more than just security (and maintenance).
- Another common aspect is **licensing**, generally you want to verify that:
 - All your dependencies *have a license* (because copyright default is “all rights reserved”).
 - All your dependency *licenses are mutually compatible*.
 - All your dependency *licenses are compatible with your licensing strategy*.
- **cargo deny**¹⁰ is yet another Cargo plugin command that provides **customizable auditing of vulnerability and licensing** aspects of your supply chain. For example:
 - For vulnerabilities: you can specify severity filters and ignore patterns.
 - For licensing: you can determine which licenses you accept, one by one or by categories (e.g., FSF-approved, OSI-approved, copyleft, permissive, etc.)
 - More generally: you can veto specific packages, unmaintained ones, etc.
- Your **deny policy** gets shipped with your code via a **deny.toml** file.
- Pro: much more flexible than **audit**.
Con: much more verbose by default too, requires careful tuning.

¹⁰<https://embarkstudios.github.io/cargo-deny/>

- Ultimately, to maximize trust in your supply chain, you need to **manually audit all the code** in it, both yours and 3rd party.
- Good news: open source is fully auditable! Only a couple of problems remain:
 1. Do you have enough *time/energy* to audit all this code?
 2. How do you keep track of *what to audit*? (In particular over time, with new releases.)
- **cargo vet**¹¹ is a Cargo command plugin by Mozilla, meant to help with (2) (and partially also with (1), via crowdsourcing).

¹¹<https://mozilla.github.io/cargo-vet/>. Introductory article: <https://lwn.net/Articles/897435/>

- Keep track in `supply-chain/audits.toml` of what/by-whom has been audited:

```
[[audits.left-pad]]
version = "1.0" # flexible version ranges/patterns are supported too
who = "Alice TheAuditor <NothingGetsPastMe@example.com>"
criteria = "safe-to-deploy"
```

- You decide your own audit criteria! Two predefined judgments: `safe-to-{run,deploy}`.
- `cargo vet`: automatically check which dependencies are not audited (or exempted).
- Supports an **interactive workflow** to: find audit targets, audit them, record outcome.
- Also help with goal (2) (minimizing effort) via **crowdsourcing**:
 - You can selectively import audits published by others:

```
[imports.foo]
url = "https://raw.githubusercontent.com/foo-team/foo/main/supply-chain/audits.toml"
[imports.bar]
url = "https://hg.bar.org/repo/raw-file/tip/supply-chain/audits.toml"
```

- Beware though: trust is not transitive!

Takeaways

- Free/Open Source Software is nowadays everywhere in IT products. With its increased prominence, it is becoming more common to encounter vulnerabilities in open source code.
- Due to its development model and dependency entanglement, a new class of attacks is getting traction: open source software supply chain attacks.
- These attacks are not something other Rust features we have seen in this course (e.g., the type system) will defend you against.
- State-of-the-art defenses are based on knowing your software dependencies and periodically audit them against known security vulnerabilities and other quality issues.
- Rust tools that help you in doing that are: `cargo {tree,audit,auditable,deny,vet}` together with the RustSec advisory database.