Security Assessment of Friktion Portfolio Management Smart Contracts

Findings and Recommendations Report Presented to:

Friktion Labs, Inc.

May 31, 2022

Version: 2.0

Presented by:

Kudelski Security, Inc.
5090 North 40th Street, Suite 450
Phoenix, Arizona 85018

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

EXECUTIVE SUMMARY

Overview

Friktion Labs, Inc. engaged Kudelski Security to perform a Security Assessment of the Friktion Portfolio Management Smart Contracts which aim to generate income for users based on options trading on various assets.

The assessment was conducted remotely by the Kudelski Security Team and our partner BTBlock. Testing took place on March 21 - April 14, 2022, and focused on the following objectives:

- Provide the customer with an assessment of their overall security posture and any risks that were discovered within the environment during the engagement.
- To provide a professional opinion on the maturity, adequacy, and efficiency of the security measures that are in place.
- To identify potential issues and include improvement recommendations based on the result of our tests.

This report summarizes the engagement, tests performed, and findings. It also contains detailed descriptions of the discovered vulnerabilities, steps the Kudelski Security Teams took to identify and validate each issue, as well as any applicable recommendations for remediation.

Key Findings

No findings of critical or high severity were found during the review.

During the test, the following positive observations were noted regarding the scope of the engagement:

- The team was very supportive and open to discussing the design choices made

Based on account relationship graph analysis and formal verification we conclude that the reviewed code implements the documented functionality.

Scope and Rules of Engagement

Kudelski performed a Security Assessment of the Friktion Portfolio Management Smart Contracts. The following table documents the targets in scope for the engagement. No additional systems or resources were in scope for this assessment.

The source code was supplied through a private repository at https://github.com/Friktion-Labs/volt with the commit hash 29335185457a912a2cda0c403a5cac08f777bb60. A re-review was performed on May 19, 2022, with the commit hash 18cc38ca136986ff08aa50b84990bfe52b24025e.

TECHNICAL ANALYSIS & FINDINGS

During the Security Assessment of the Friktion Portfolio Management Smart Contracts, we discovered:

- 3 findings with LOW severity rating.
- 8 findings with INFORMATIONAL severity rating.

The following chart displays the findings by severity.

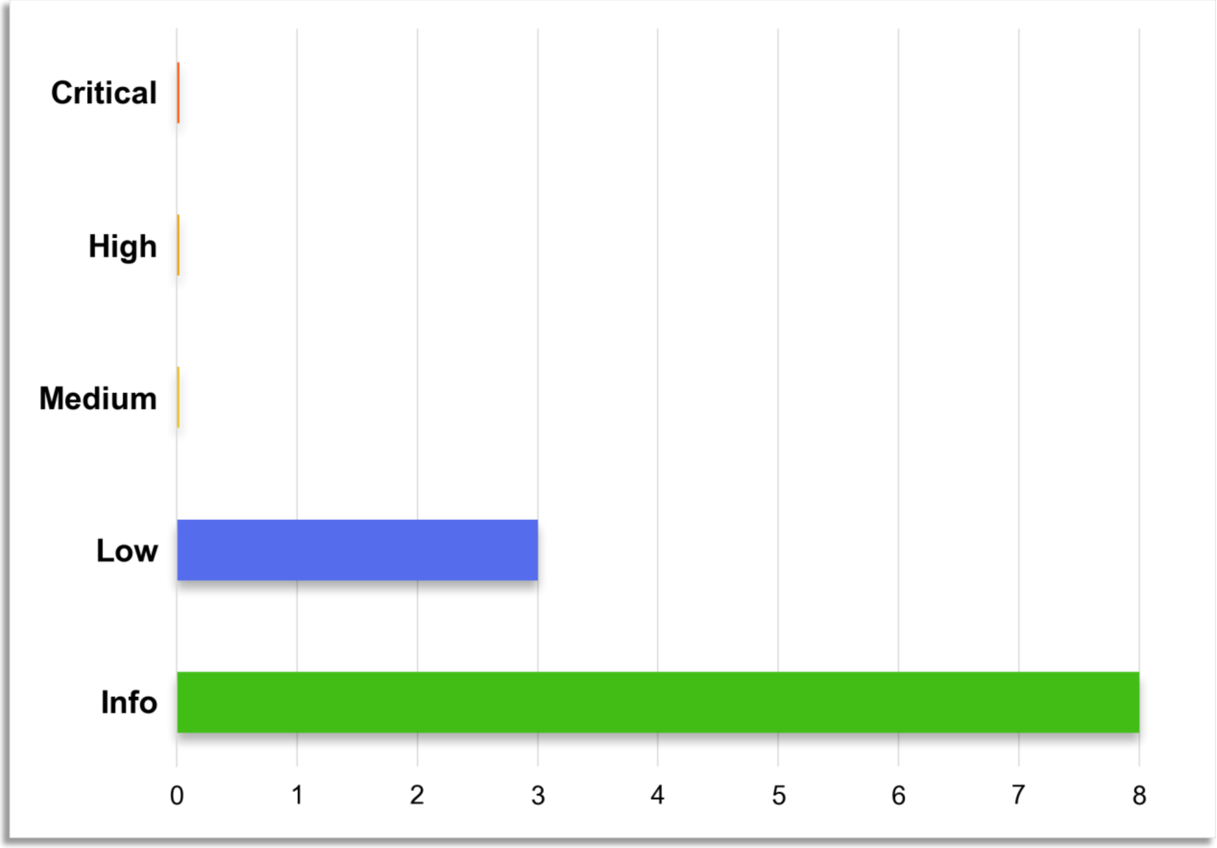

Figure 1: Findings by Severity

Findings

The Findings section provides detailed information on each of the findings, including methods of discovery, explanation of severity determination, recommendations, and applicable references.

The following table provides an overview of the findings.

#	Severity	Description
KS-FRIKTION-01	Low	Match statement error
KS-FRIKTION-02	Low	No check of authority on whitelist token mint
KS-FRIKTION-03	Low	No check on oracle account
KS-FRIKTION-04	Informational	Debug message vs. code discrepancy
KS-FRIKTION-05	Informational	Duplicate checks
KS-FRIKTION-06	Informational	Issue with a debug print
KS-FRIKTION-07	Informational	Misleading comments
KS-FRIKTION-08	Informational	Use UncheckedAccount over AccountInfo
KS-FRIKTION-09	Informational	Use of access control for custom errors
KS-FRIKTION-10	Informational	Use of init_if_needed without need
KS-FRIKTION-11	Informational	Using ///CHECK without any explanation to silence anchor build

Table 1: Findings Overview

Technical Analysis

The source code has been manually validated to the extent that the state of the repository allowed. The validation includes confirming that the code correctly implements the intended functionality.

Further investigations concluded that no critical risks were identified for the application, including:

- No potential panics were detected
- No potential errors regarding wraps/unwraps, expect and wildcards
- No internal unintentional unsafe references

Authorization

The review used relationship graphs to show the relations between account input passed to the instructions of the program. The relations are used to verify if the authorization is sufficient for invoking each instruction. The graphs show if any unreferenced accounts exist. Accounts that are not referred to by trusted accounts can be replaced by any account of an attacker's choosing and thus pose a security risk.

In particular, the graphs will show if signing accounts are referred to. If a signing account is not referred to then any account can be used to sign the transaction causing insufficient authorization.

No insufficient authorization was found based on the analyzes of the relationship graphs. For details, see section **Error! Reference source not found.** starting on page **Error! Bookmark not defined.**.

Conclusion

Based on account relationship graph analysis and formal verification we conclude that the code implements the documented functionality to the extent of the reviewed code.

Technical Findings

General Observations

During code assessment, it was noted that the Rust code is well written and the use of checked arithmetic operations to protect from overflow/underflow operations combined with `vipers` `unwrap_int` shows commitment to writing secure programs.

The code documentation is okay, however doc comments required by anchor were mostly just silenced instead of explaining why account checks are not required. There are also a lot of copy and paste comments that have not been properly modified to reflect the actual code. Examples are false statements about what token is burned and whether a function can run before or after expiry. There are quite a few more instances of such comment related issues. It may also be a good idea to refrain from the use of swear words in comments even if used in jest.

The use of the anchor framework with its inbuilt account verification functionality is a great foundation for this Solana project. The programs strongly rely on anchors inbuilt validation and access restriction macros. Though many of the checks have been duplicated, a few of them appear 3 times. This is due to checks being implemented in account macros, access control functions as well as the handler function. While Solana, unlike Ethereum, does not charge gas fees based on each instruction, the problem with this is maintainability and readability of the code. Many of the access control functions are complete duplicates of checks already done in the account macros, thus the whole access control function could be removed. Custom errors can also be used in account macros.

Match statement error

Finding ID: KS-FRIKTION-01
Severity: **Low**
Status: **Remediated**

Description

A defective match statement that always executes the first branch. This will lead to miscalculated fees for the Inertia protocol.

Proof of Issue

File name: programs/volt/src/helpers.rs
Line number: 542

```
match self.options_protocol_name {
    Soloptions => soloptions::fees::calculate_mint_fee(ul_amount).unwrap(),
    Inertia => inertia::fees::mint_fee_amount(ul_amount),
}
```

This match statement does not match on Types, it assigns the Enum to the variable called `Soloptions` and always executes the first branch.

Severity and Impact Summary

When calculating the fee for `OptionsProtocol::Inertia` it will instead return the fee for `OptionsProtocol::Soloptions`. Therefore, the wrong fee is always charged when the Inertia protocol is used.

Recommendation

Match on the actual Type.

```
match self.options_protocol_name {
    OptionsProtocol::Soloptions =>
soloptions::fees::calculate_mint_fee(ul_amount).unwrap(),
    OptionsProtocol::Inertia => inertia::fees::mint_fee_amount(ul_amount),
}
```

No check of authority on whitelist token mint

Finding ID: KS-FRIKTION-02
Severity: **Low**
Status: **Remediated**

Description

The authority over the provided mint account is not checked. Whitelist tokens grant access and should therefore be issued by a trusted party. While this is an admin function, it is still a good idea to verify the mint authority to prevent any misuse.

Proof of Issue

File name: programs/volt/src/ixs/dov/initialize.rs
Line number: 121

```
pub whitelist_token_mint: Box<Account<'info, Mint>>,
```

Severity and Impact Summary

This is an “admin function” creating a new volt vault. For the `whitelist_token_mint` account it is established that it is a `Mint` account and therefore owned by the token program. But it is not checked who is authorized to mint those whitelist tokens. The admin could pass in any authority – with ill intention or by mistake.

Recommendation

It is recommended to restrict the authority of the mint account even though the person executing this function is trusted.

No check on oracle account

Finding ID: KS-FRIKTION-03
Severity: **Low**
Status: **Remediated**

Description

There is no account check for the oracle_ai account passed in `inertia::new_contract()`.

Proof of Issue

File name: programs/inertia/src/contexts.rs
Line number: 20

```
/// CHECK: difficult to authenticate which oracle protocol owns this pubkey through
anchor macros.
pub oracle_ai: AccountInfo<'info>,
```

Severity and Impact Summary

This is an "admin function" creating a new options contract and the account key given here is stored for future reference. Other program functions compare the key to the account given & stored here. Therefore, the account cannot be changed later, and account validation has to be done here. It may be prudent to make sure that this account is indeed an oracle account, that it is owned by the relevant program.
If, by mistake or intent, some other account is passed in, the `option_settle()` function could become un-executable as it attempts to de-serialize this account. This would, in turn, also make the `option_exercise()` and `option_redeem()` functions inaccessible. Account de-serialization is attempted in `AggregatorAccountData::new()` and requires a discriminator to match.

```
let temp_decimal: f64 = AggregatorAccountData::new(oracle_ai)?
  .get_result()?
  .try_into()
  .unwrap();
```

Recommendation

It is recommended to check the ownership of this account using anchors `Account` type or `owner` account macro and/or attempt to de-serialize this account in `new_contract()` to make sure that it won't fail later when `option_settle()` is invoked.

References

https://docs.rs/anchor-lang/latest/anchor_lang/derive.Accounts.html
https://docs.switchboard.xyz/program#mainnet-beta

Debug message vs. code discrepancy

Finding ID: KS-FRIKTION-04
Severity: **Informational**
Status: **Remediated**

Description

Debug message states the required amount should be greater than 0. Code checks it is 0.

Proof of Issue

File name: programs/volt/src/ixs/dov/settle_permissioned_market_premium_funds.rs
Line number: 97

```
msg!("checking require amount > 0");

require!(
    ctx.accounts.permissioned_market_premium_pool.amount == 0,
    InvalidPermissionedMarketPremiumMint
)
```

Severity and Impact Summary

No security impact.

Recommendation

It could be a good idea to keep the debug messages coherent with the actual code.

Duplicate checks

Finding ID: KS-FRIKTION-05
Severity: **Informational**
Status: **Open**

Description

There are about 100 duplicate checks. These checks are often implemented on the account and again in the access control function.

Proof of Issue

File name: programs/volt/src/ixs/dov/rebalance_swap_premium.rs
Line number: 43

```
#[account(mut, address=volt_vault.vault_authority)]
/// CHECK: skip
pub vault_authority: AccountInfo<'info>,
```

And: **File name:** programs/volt/src/account_validators.rs
Line number: 382

```
if self.vault_authority.key() != self.volt_vault.vault_authority {
  return Err(VaultAuthorityDoesNotMatch.into());
}
```

Severity and Impact Summary

The unnecessary duplication of checks bloats the code and makes it more difficult to understand. Often all the checks in the access control function are duplicates, thus the whole access control function could be removed.

Recommendation

Keep the code concise, adding an access control function should only be done if the checks cannot be expressed in the account macros.

Issue with a debug print

Finding ID: KS-FRIKTION-06
Severity: **Informational**
Status: **Remediated**

Description

The debug print will print the same accounts for before and after.

Proof of Issue

File name: programs/volt/src/ixs/whitelist/remove_whitelist.rs
Line number: 28

```
msg!("addresses before: {:?}", ctx.accounts.whitelist.addresses);
let key_to_remove = ctx.accounts.account_to_remove.key();
msg!("addresses after: {:?}", ctx.accounts.whitelist.addresses);
ctx.accounts
  .whitelist
  .addresses
  .retain(|x| *x != key_to_remove);
```

Severity and Impact Summary

The order of the debug prints is wrong.

Recommendation

Move the print function beneath the call to `retain()`.

Misleading comments

Finding ID: KS-FRIKTION-07
Severity: **Informational**
Status: **Remediated**

Description

In `close_position()` there is a misleading comment stating that positions can only be closed after expiry, while the code implements a check to verify that the contract is not yet expired. And there are two burn function calls: one to burn option tokens and one for writer tokens. However, both functions have a comment that states that writer tokens are burned. Presumably that comment is a copy and paste.

Proof of Issue

File name: programs/inertia/src/lib.rs
Line number: 187

```
// dont exercise before expiry
require!(
  (ctx.accounts.clock.unix_timestamp as u64) < contract.expiry_ts,
  TooLateToClosePosition
);
```

File name: programs/inertia/src/lib.rs
Line number: 237

```
// burn writer tokens
token::burn(
    CpiContext::new(
        ctx.accounts.token_program.to_account_info(),
        token::Burn {
            mint: ctx.accounts.option_mint.to_account_info(),
 ...
```

Severity and Impact Summary

There are misleading comments throughout the project which may briefly confuse future maintainers of the project.

Recommendation

It would be good to have accurate comments.

Use UncheckedAccount over AccountInfo

Finding ID: KS-FRIKTION-08
Severity: **Informational**
Status: **Open**

Description

AccountInfo is always used instead of UncheckedAccount

Proof of Issue

File name: initialize.rs
Line number: 27

```
/// CHECK: skip
admin_key: AccountInfo<'info>,
```

And many other places.

Severity and Impact Summary

The code would be more in line with best practices if it were to follow the guidelines set by Anchor.

Recommendation

Anchor recommends the use of UncheckedAccount over AccountInfo to emphasize the lack of checks made.

References

https://docs.rs/anchor-lang/latest/anchor_lang/accounts/account_info/index.html

Use of access control for custom errors

Finding ID: KS-FRIKTION-09
Severity: **Informational**
Status: **Open**

Description

The use of custom error messages is great, but it is not necessary to use access control functions for this purpose.

Proof of Issue

File name: programs/volt/src/account_validators.rs
Line number: 448

```
// Note: This is an unncessary check as the signer seeds to mint the vault tokens
would
//  fail, but this procvides a nice custom error message
if self.vault_authority.key() != self.volt_vault.vault_authority {
  return Err(VaultAuthorityDoesNotMatch.into());
}
```

Severity and Impact Summary

Custom errors can be added to the account macro-based checks, thus making the code more concise:

```
#[account(address = <expr> @ <custom_error>)]
```

Recommendation

Keep the code concise by adding the custom errors to the account macro-based checks.

References

https://docs.rs/anchor-lang/latest/anchor_lang/derive.Accounts.html

Use of init_if_needed without need

Finding ID: KS-FRIKTION-10
Severity: **Informational**
Status: **Remediated**

Description

The use of `init_if_needed` is discouraged by anchor and was therefore put behind a feature flag. While `init_if_needed` is generally not that much of an issue with PDAs, it may be better to avoid its use if not required.

Proof of Issue

File name: programs/volt/src/ixs/template/cancel_pending_withdrawal.rs
Line number: 35

```
#[account(init_if_needed,
    seeds = [volt_vault.key().as_ref(), authority.key().as_ref(),
b"pendingWithdrawal"],
    bump,
    payer = authority)]
pub pending_withdrawal_info: Box<Account<'info, PendingWithdrawal>>,
```

Severity and Impact Summary

There is no security issue with the code as the account is a PDA and does have an `is initialized` check. However, in this two-step process, a withdrawal has to be initialized before it could be cancelled. This means there is no need to initialize the account in the cancel function.

Line number: 57

```
// Validate that pending_withdrawal_info is initialized
require!(
    ctx.accounts.pending_withdrawal_info.initialized,
    PendingWithdrawalInfoNotInitialized
);
```

Recommendation

Avoid the use of `init_if_needed` if not actually required. Another can be found in `cancel_pending_deposit` for the `pending_deposit_info` acccount.
Since the `initialize` and `initialize_entropy` functions setup the `extra_volt_data` account, it may be worth checking if the `init_if_needed` is (still) relevant for `withdraw`, `deposit_dao`, `attach_whitelist`.

References

https://docs.rs/anchor-lang/latest/anchor_lang/derive.Accounts.html#normal-constraints

Using ///CHECK without any explanation to silence anchor build

Finding ID: KS-FRIKTION-11
Severity: **Informational**
Status: **Remediated**

Description

Anchor requires that the use of unchecked accounts is annotated with a doc comment elaborating the reason why checks are not required.
The most common explanation given in the audited programs is “skip”. This is not an explanation.

Proof of Issue

File name: initialize.rs
Line number: 26

```rust
/// CHECK: skip
admin_key: AccountInfo<'info>,
```

Severity and Impact Summary

The explanation “skip” does not actually describe why no checks are required. While it does silence `anchor build` it does not help new developers understand the reasoning.
There are 280 doc comments “CHECK: skip”.

Recommendation

It could be helpful during the maintenance of the programs if the required documentation were more helpful.

References

https://book.anchor-lang.com/chapter_3/the_accounts_struct.html#safety-checks

METHODOLOGY

Kudelski Security uses the following high-level methodology when approaching engagements. They are broken up into the following phases.

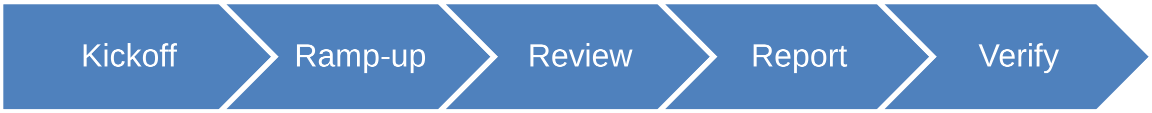

Figure 2: Methodology Flow

Kickoff

The project is kicked off as the sales process has concluded. We typically set up a kickoff meeting where project stakeholders are gathered to discuss the project as well as the responsibilities of participants. During this meeting we verify the scope of the engagement and discuss the project activities. It's an opportunity for both sides to ask questions and get to know each other. By the end of the kickoff there is an understanding of the following:

- Designated points of contact
- Communication methods and frequency
- Shared documentation
- Code and/or any other artifacts necessary for project success
- Follow-up meeting schedule, such as a technical walkthrough
- Understanding of timeline and duration

Ramp-up

Ramp-up consists of the activities necessary to gain proficiency on the particular project. This can include the steps needed for familiarity with the codebase or technological innovation utilized. This may include, but is not limited to:

- Reviewing previous work in the area including academic papers
- Reviewing programming language constructs for specific languages
- Researching common flaws and recent technological advancements

Review

The review phase is where most of the work on the engagement is completed. This is the phase where we analyze the project for flaws and issues that impact the security posture. Depending on the project this may include an analysis of the architecture, a review of the code, and a specification matching to match the architecture to the implemented code.

In this code audit, we performed the following tasks:

1. Security analysis and architecture review of the original protocol

2. Review of the code written for the project
3. Compliance of the code with the provided technical documentation

The review for this project was performed using manual methods and utilizing the experience of the reviewer. No dynamic testing was performed, only the use of custom-built scripts and tools were used to assist the reviewer during the testing. We discuss our methodology in more detail in the following sections.

Code Safety

We analyzed the provided code, checking for issues related to the following categories:

- General code safety and susceptibility to known issues
- Poor coding practices and unsafe behavior
- Leakage of secrets or other sensitive data through memory mismanagement
- Susceptibility to misuse and system errors
- Error management and logging

This list is general list and not comprehensive, meant only to give an understanding of the issues we are looking for.

Technical Specification Matching

We analyzed the provided documentation and checked that the code matches the specification. We checked for things such as:

- Proper implementation of the documented protocol phases
- Proper error handling
- Adherence to the protocol logical description

Reporting

Kudelski Security delivers a preliminary report in PDF format that contains an executive summary, technical details, and observations about the project.

The executive summary contains an overview of the engagement including the number of findings as well as a statement about our general risk assessment of the project. We may conclude that the overall risk is low but depending on what was assessed we may conclude that more scrutiny of the project is needed.

We not only report security issues identified but also informational findings for improvement categorized into several buckets:

- Critical
- High
- Medium
- Low
- Informational

The technical details are aimed more at developers, describing the issues, the severity ranking and recommendations for mitigation.

As we perform the audit, we may identify issues that aren't security related, but are general best practices and steps, that can be taken to lower the attack surface of the project. We will call those out as we encounter them and as time permits.

As an optional step, we can agree on the creation of a public report that can be shared and distributed with a larger audience.

Verify

After the preliminary findings have been delivered, this could be in the form of the approved communication channel or delivery of the draft report, we will verify any fixes withing a window of time specified in the project. After the fixes have been verified, we will change the status of the finding in the report from open to remediated.

The output of this phase will be a final report with any mitigated findings noted.

Additional Note

It is important to note that, although we did our best in our analysis, no code audit or assessment is a guarantee of the absence of flaws. Our effort was constrained by resource and time limits along with the scope of the agreement.

While assessment the severity of the findings, we considered the impact, ease of exploitability, and the probability of attack. These is a solid baseline for severity determination.

The Classification of identified problems and vulnerabilities

There are four severity levels of an identified security vulnerability.

Critical – vulnerability that will lead to loss of protected assets

- This is a vulnerability that would lead to immediate loss of protected assets
- The complexity to exploit is low
- The probability of exploit is high

High - A vulnerability that can lead to loss of protected assets

- All discrepancies found where there is a security claim made in the documentation that cannot be found in the code
- All mismatches from the stated and actual functionality
- Unprotected key material
- Weak encryption of keys
- Badly generated key materials

- Tx signatures not verified
- Spending of funds through logic errors
- Calculation errors overflows and underflows

Medium - a vulnerability that hampers the uptime of the system or can lead to other problems

- Insecure calls to third party libraries
- Use of untested or nonstandard or non-peer-revied crypto functions
- Program crashes leaves core dumps or write sensitive data to log files

Low - Problems that have a security impact but does not directly impact the protected assets

- Overly complex functions
- Unchecked return values from 3rd party libraries that could alter the execution flow

Informational

- General recommendations

Tools

The following tools were used during this portion of the test. A link for more information about the tool is provided as well.

Tools used during the code review and assessment

- Rust – cargo tools
- IDE modules for Rust and analysis of source code
- Cargo audit which uses https://rustsec.org/advisories/ to find vulnerabilities cargo.

RustSec.org

About RustSec

The RustSec Advisory Database is a repository of security advisories filed against Rust crates published and maintained by the Rust Secure Code Working Group.

The RustSec Tool-set used in projects and CI/CD pipelines

- `cargo-audit` - audit Cargo.lock files for crates with security vulnerabilities.
- `cargo-deny` - audit Cargo.lock files for crates with security vulnerabilities, limit the usage of particular dependencies, their licenses, sources to download from, detect multiple versions of same packages in the dependency tree and more.