Synthetify Platform Code Audit and Architecture Review

Findings and Recommendations Report Presented to:

Synthetify

October 11, 2021
Version: 1.0 Final

Presented by:

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

FOR PUBLIC RELEASE

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

EXECUTIVE SUMMARY

Overview

Synthetify engaged Kudelski Security to perform a Synthetify Platform Code Audit and Architecture Review.

The assessment was conducted remotely by the Kudelski Security Team. Testing took place on August 30 – September 23, 2021, 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.

The re-review of the findings were concluded on October 9, 2021. ***At that point all findings had been remediated.***

Key Findings

The following are the major themes and issues identified during the testing period. These, along with other items, within the findings section, are the most important items to prioritize for remediation to reduce to the risk they pose.

- Extensive use of unwrap calls
- Add constraints to InitializeAssetsList
- Use of unwrap is system critical functions

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

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

Based on the account relationship graphs or reference graphs and the formal verification we can conclude that the reviewed code implements the documented functionality.

Scope and Rules Of Engagement

Kudelski performed an Synthetify Platform Code Audit and Architecture Review for Synthetify. 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 private repository at https://github.com/Synthetify/synthetify-protocol with the commit hash 3f161431499eb38fb381956fc35406be365571cd

The source code for the re-review was supplied through the now public repository at https://github.com/Synthetify/synthetify-protocol with the commit hash de5a26bae7bae5e4fec0c526ac86b9429d0c54a1

Files included in the code review	
synthetify-protocol/ ├── docs ├── programs │ ├── exchange │ │ ├── Cargo.toml │ │ ├── src │ │ │ ├── account.rs │ │ │ ├── context.rs │ │ │ ├── decimal.rs │ │ │ ├── lib.rs │ │ │ ├── math.rs │ │ │ └── utils.rs │ │ └── Xargo.toml │ └── pyth │ ├── Cargo.toml │ ├── src │ │ ├── lib.rs │ │ └── pc.rs │ └── Xargo.toml │	Rust implementation of the program and documentation

Table 1: Scope

TECHNICAL ANALYSIS & FINDINGS

During the Synthetify Platform Code Audit and Architecture Review, we discovered 3 findings that had a MEDIUM severity rating, as well as 1 LOW

The following chart displays the findings by severity.

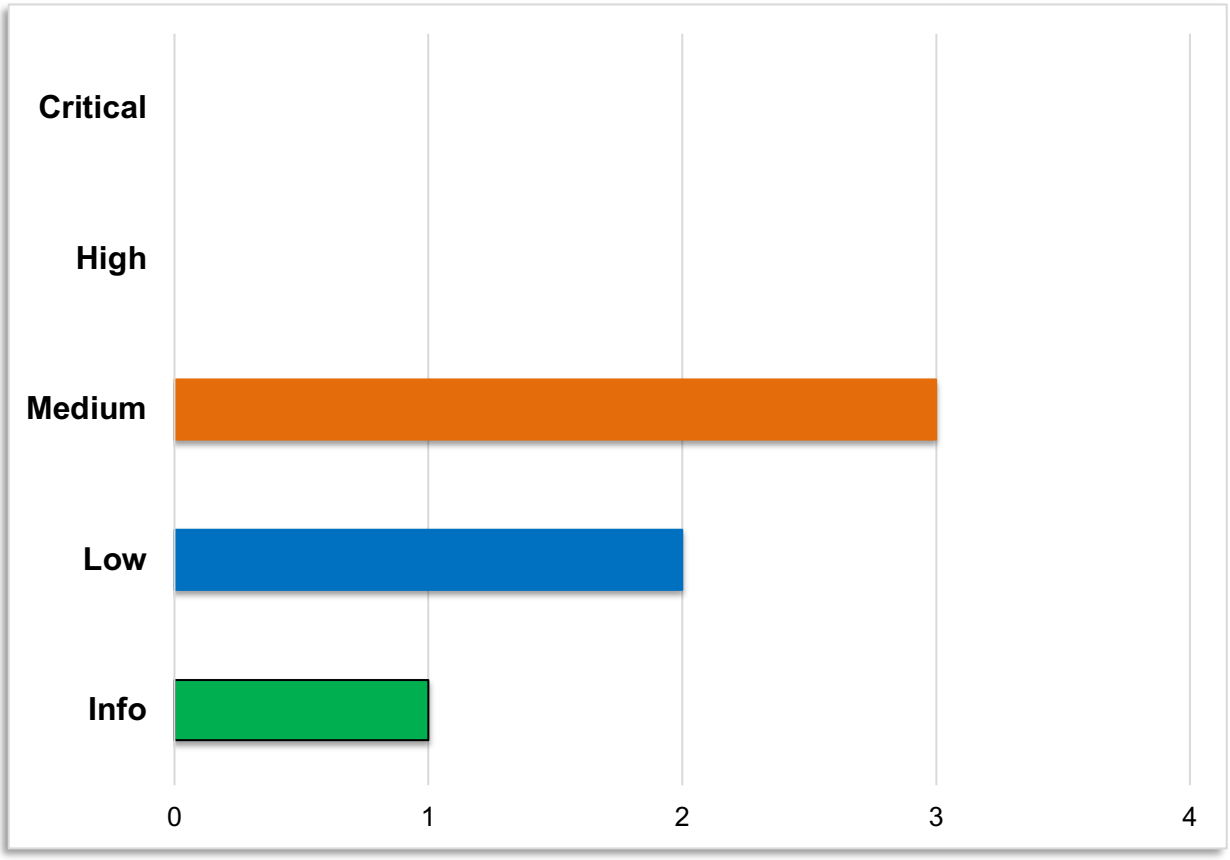


Figure 1: Findings by Severity

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-SYNTH-F-01	Medium	Extensive use of unwrap calls
KS-SYNTH-F-02	Medium	Add constraints to InitializeAssetsList
KS-SYNTH-F-03	Medium	Use of unwrap is system critical functions
KS-SYNTH-F-04	Low	Low test coverage
KS-SYNTH-F-05	Low	Outdated packages and yanked references
KS-SYNTH-F-06	Informational	Unreferenced accounts in account structs

Table 2: Findings Overview

Technical analysis

Based on the source code the following account relationship graphs or reference graphs was made to verify the validity of the code as well as comfirmating that the intended functionality was implemented correctly and to the extent that the state of the repository allowed.

A number of further investigations were made which conluded that they did not pose a risk to the application. They were

- 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.

- Input struct is `AddCollateral`
- Access control check on `state.admin==admin.key`

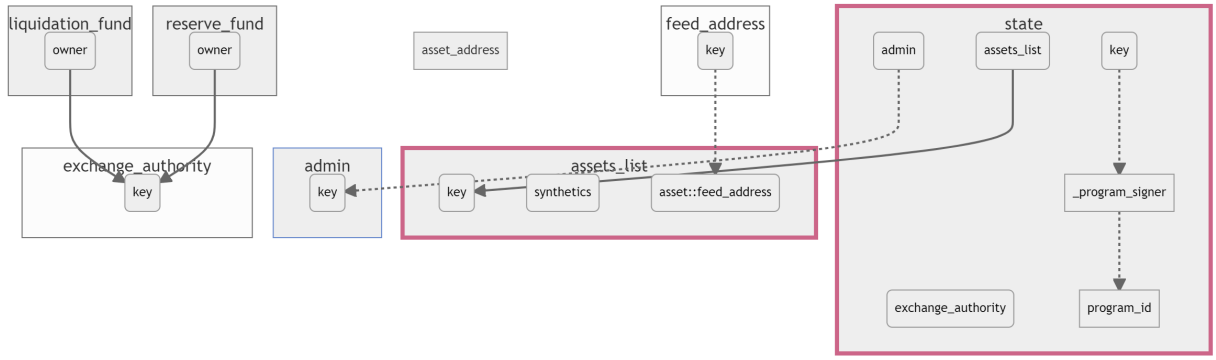


Figure 2: add_colateral

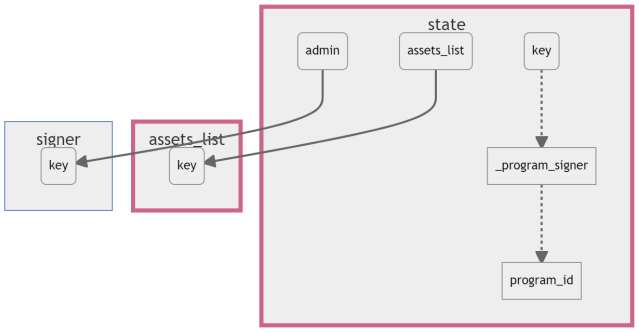


Figure 3: add_new_asset

- Input accounts struct `AddSynthetic`
- Access control check `state.admin==admin.key`

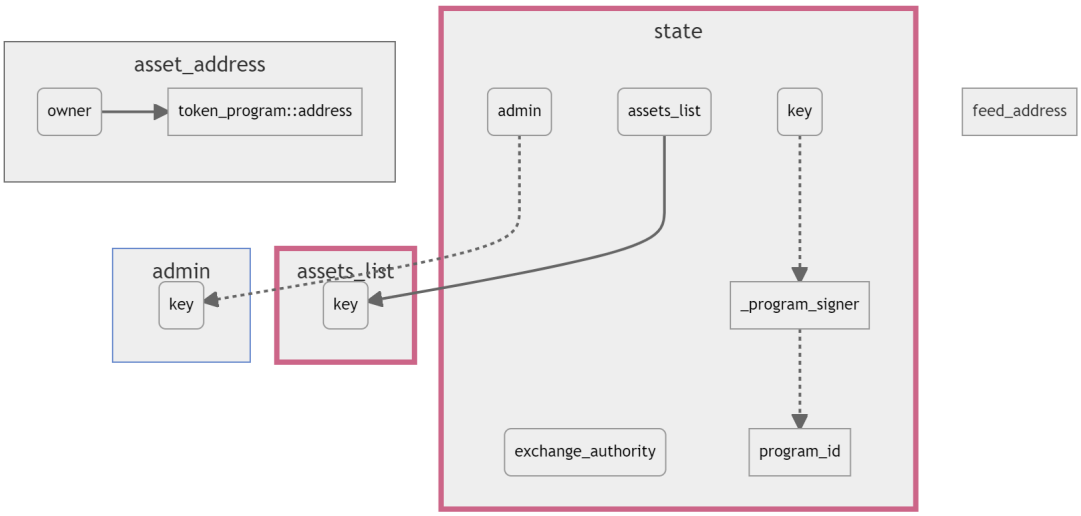


Figure 4: add_synthetic

- Input accounts struct `BorrowVault`
- Access control check on `state` halted and `vault` halted.

l1925

```
        let synthetic_position = synthetics
            .iter_mut()
            .position(|x| {
                x.asset_address
                    .eq(ctx.accounts.synthetic.to_account_info().key)
            })
            .unwrap();

        let collateral_position = collaterals
            .iter()
            .position(|x| {
                x.collateral_address
                    .eq(ctx.accounts.collateral.to_account_info().key)
            })
            .unwrap();
```

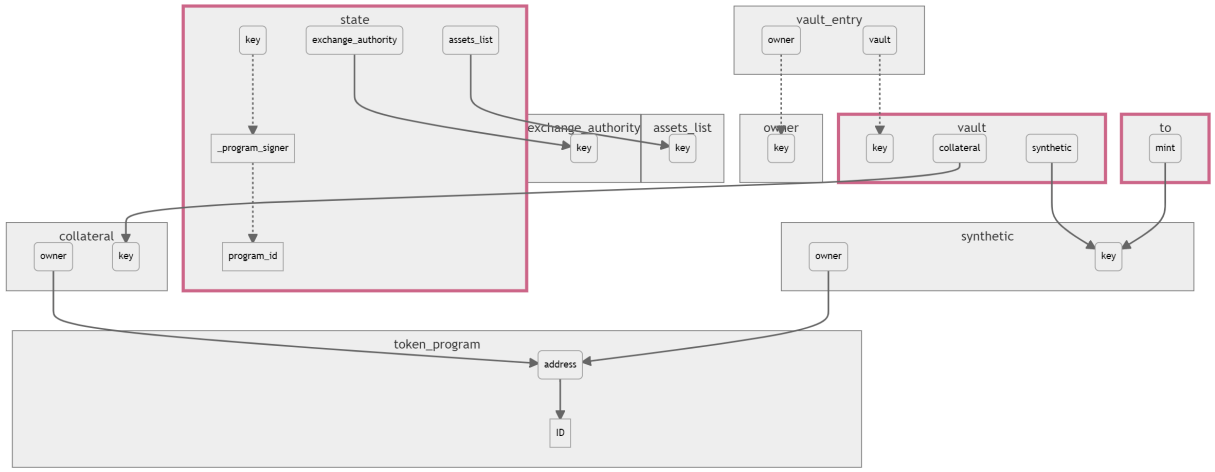


Figure 5: borrow_vault

- Input accounts struct is `BurnToken`

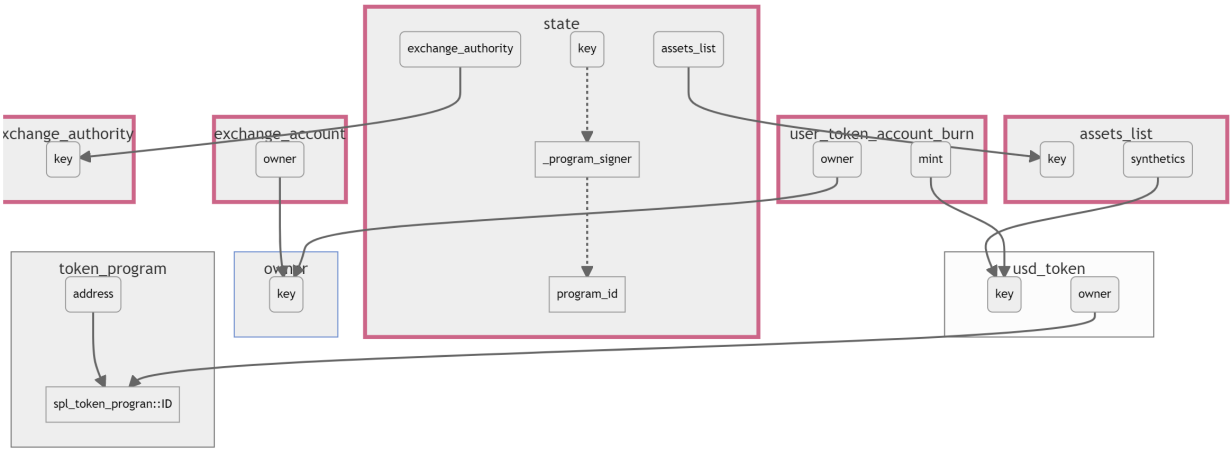


Figure 6: burn

- Access control check on `state.halted=True`
- Should the exchange owner account be forced to sign this so that we are sure that the account isn't modified?

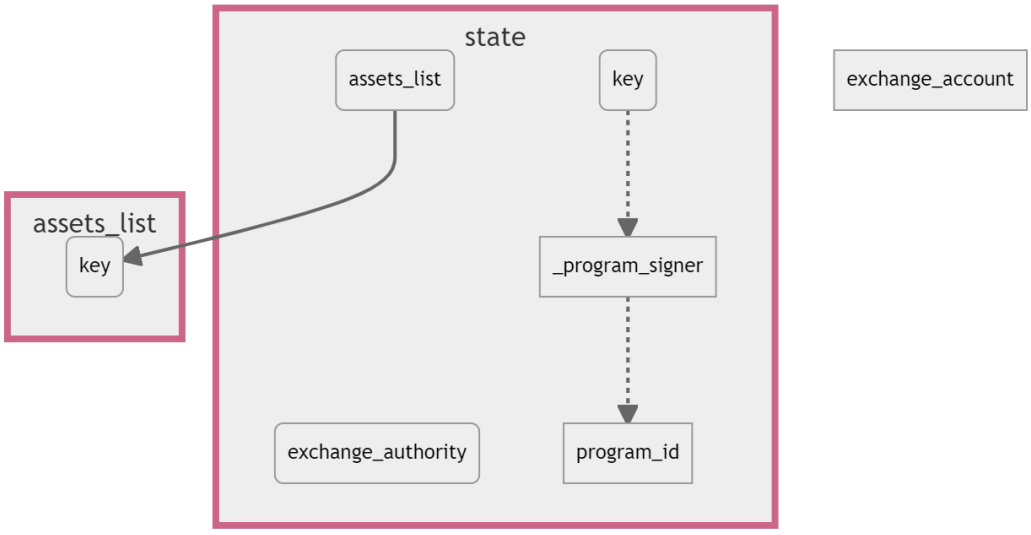


Figure 7: check_account_collateralization

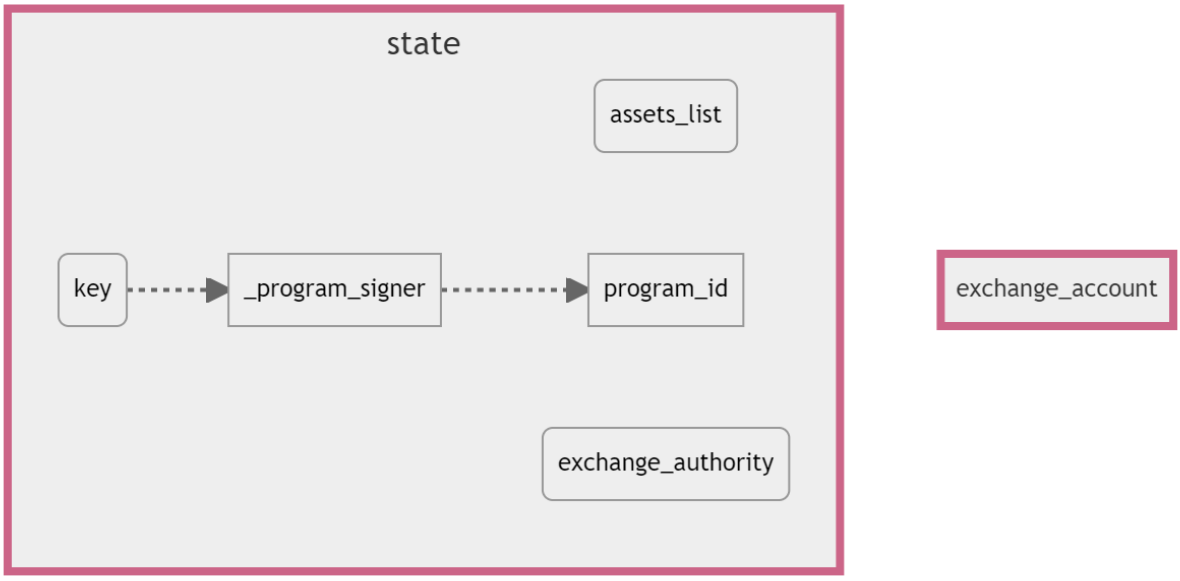


Figure 8: claim_rewards

- Input ctx struct is `CreateExchangeAccount`

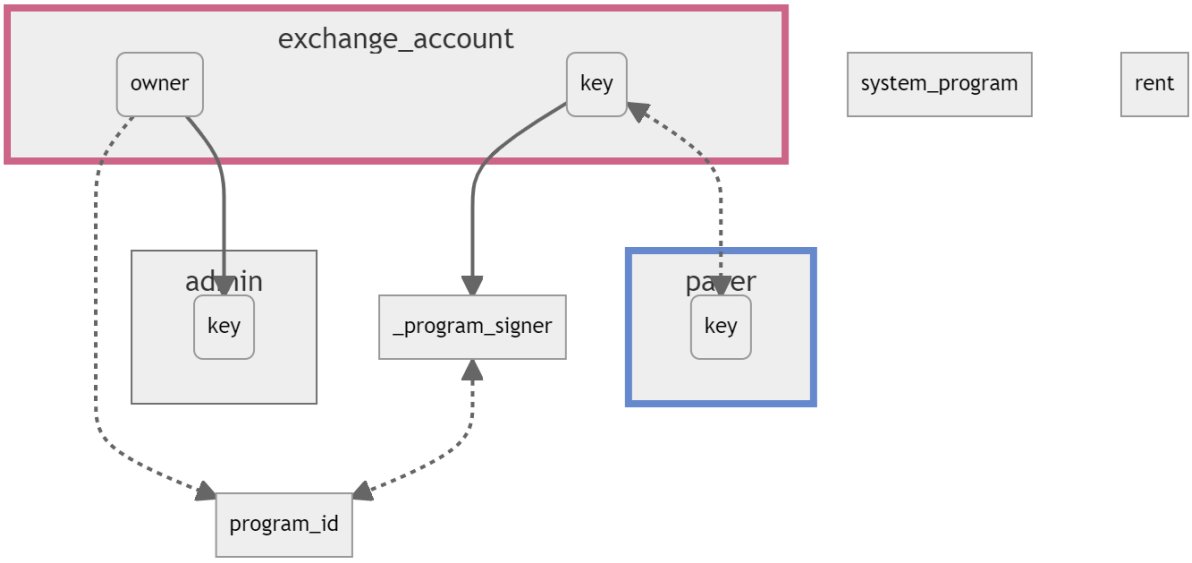


Figure 9: create_exchange_account

- Before execution access_control tag verifies that `ctx.accounts.state`, `ctx.accounts.admin.key==ctx.accounts.state.load().admin`.
- Input ctx struct is `InitializeAssetsList`.
- Three pub keys passed in as arguements (Is this a good idea? Will they be verified?)
- `sny_reserve` and `sby_liquidation_fund` are used with no verification to initialize the `AssetList`. Why does it happen this way?

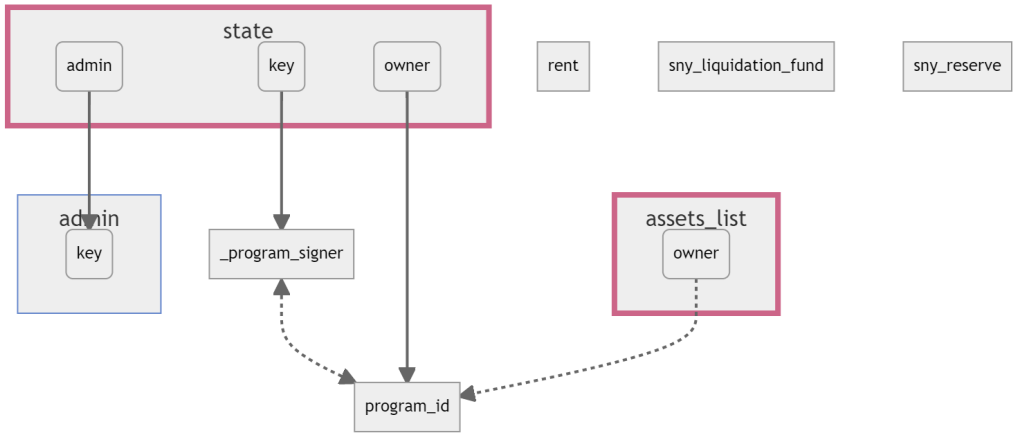


Figure 10: create_list

- Input accounts struct `CreateSwapLine`
- Access control check `state.admin==admin.key`

l1573

```
let synthetic = assets_list
    .synthetics
    .iter()
    .find(|x| {
        x.asset_address
            .eq(ctx.accounts.synthetic.to_account_info().key)
    })
    .unwrap();
```

l1581

```
let collateral = assets_list
    .collaterals
    .iter()
    .find(|x| {
        x.collateral_address
            .eq(&ctx.accounts.collateral.to_account_info().key)
    })
    .unwrap();
```

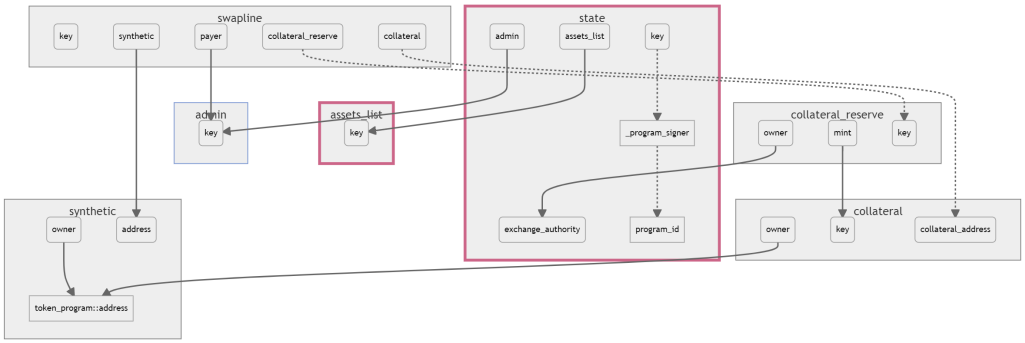


Figure 11: create_swap_line

- Access control on state halted
- Access control on `state.admin==admin.key`
- Input account struct `CreateVault`

l 1791

```
let synthetic = assets_list
    .synthetics
    .iter()
    .find(|x| {
        x.asset_address
            .eq(ctx.accounts.synthetic.to_account_info().key)
    })
    .unwrap();
```

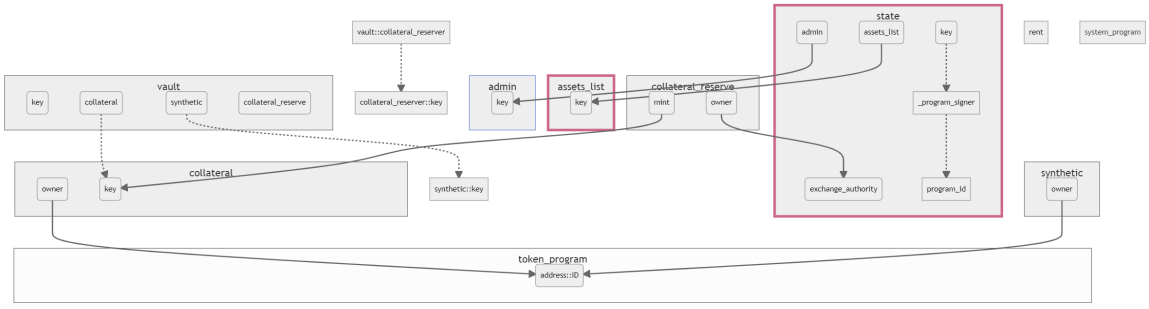


Figure 12: create_vault

- Access control on state halted
- Accesss control on vault halted
- Input structs account `CreateVaultEntry`

l1844

```
        let synthetic = assets_list
            .synthetics
            .iter()
            .find(|x| x.asset_address.eq(&vault.synthetic))
            .unwrap();
        let collateral = assets_list
            .collaterals
            .iter()
            .find(|x| x.collateral_address.eq(&vault.collateral))
            .unwrap();
```

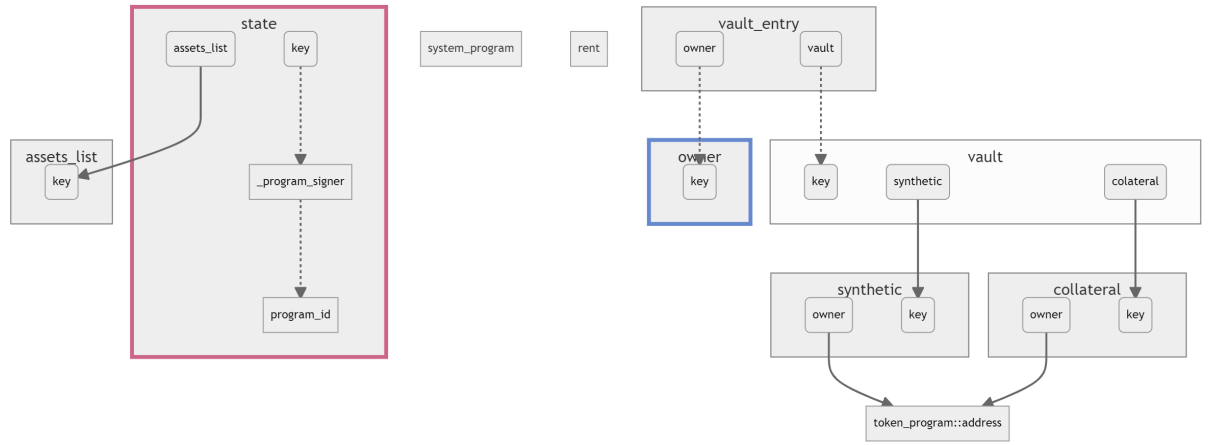


Figure 13: create_vault_entry

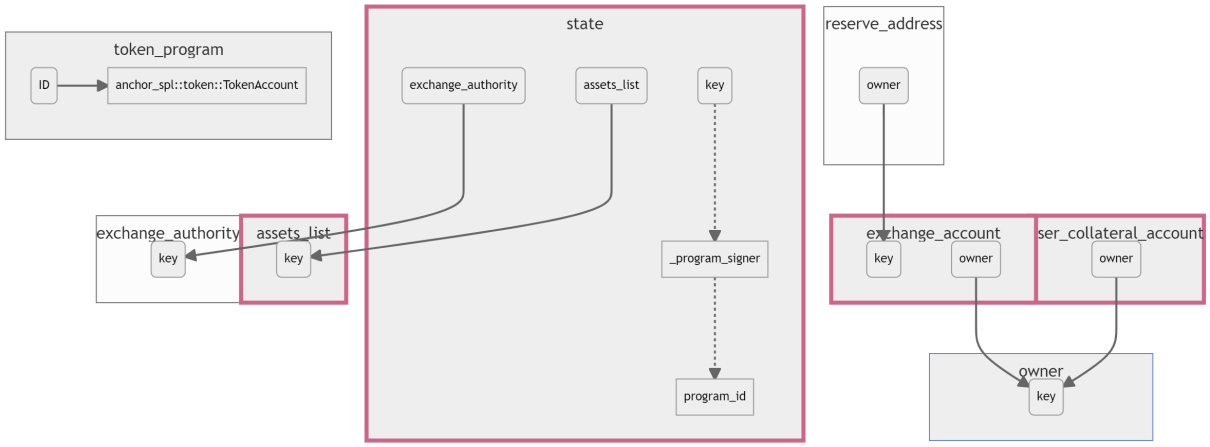


Figure 14:deposit

- Input accounts struct `DepositVault`
- Access control check on `state` halted and `vault` halted.

l 1880

```
    let collateral = collaterals
        .iter()
        .find(|x| {
            x.collateral_address
                .eq(ctx.accounts.collateral.to_account_info().key)
        })
        .unwrap();

    let synthetic = synthetics
        .iter_mut()
        .find(|x| {
            x.asset_address
                .eq(ctx.accounts.synthetic.to_account_info().key)
        })
        .unwrap();
```

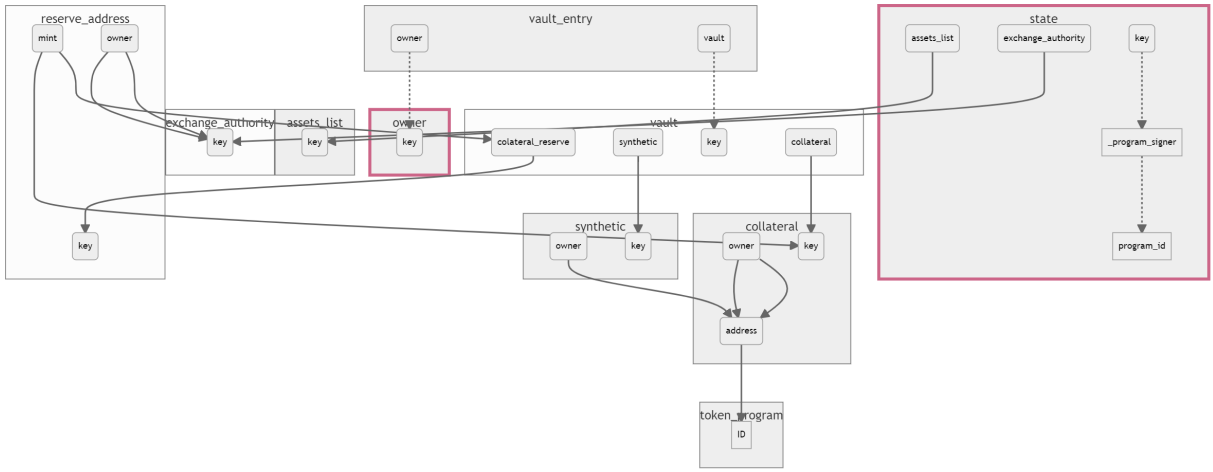


Figure 15: deposit_vault

- Input accounts struct is `Init`.
-

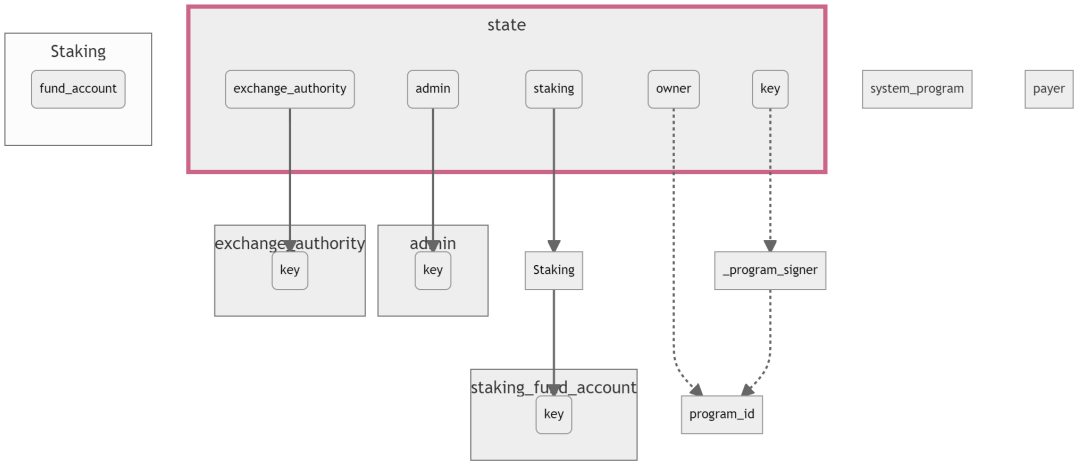


Figure 16: init

- Access control check on `state.halted=True`

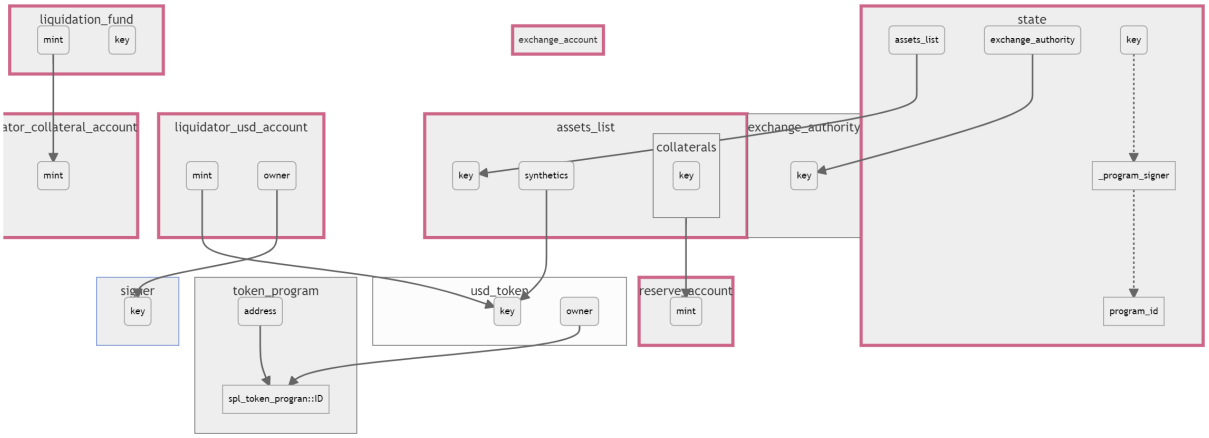


Figure 17: liquidate

- Input accounts struct `LiquidateVault`
- Access control check on `state` halted and `vault` halted.

l2153

```
        let synthetic_position = synthetics
            .iter_mut()
            .position(|x| {
                x.asset_address
                    .eq(ctx.accounts.synthetic.to_account_info().key)
            })
            .unwrap();

        let collateral_position = collaterals
            .iter()
            .position(|x| {
                x.collateral_address
                    .eq(ctx.accounts.collateral.to_account_info().key)
            })
            .unwrap();
```

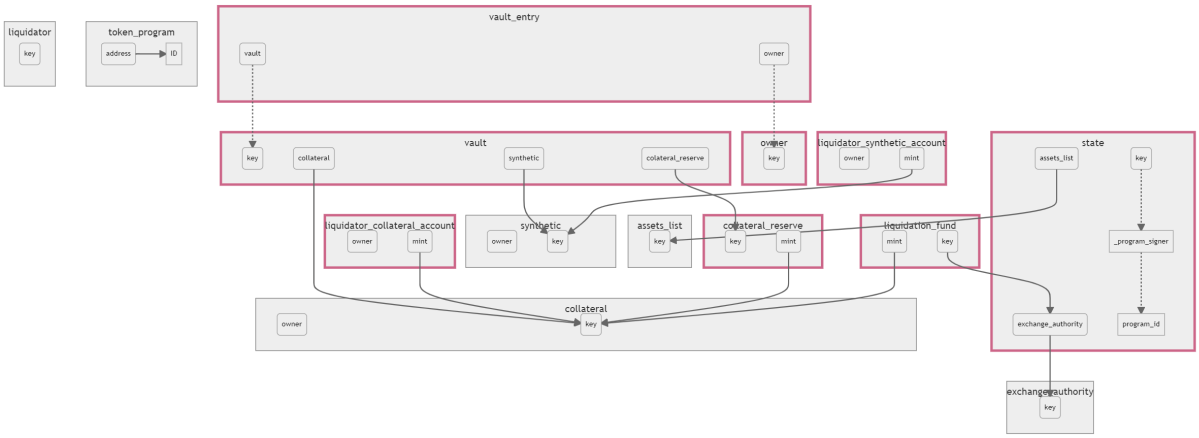


Figure 18: liduidate_vault

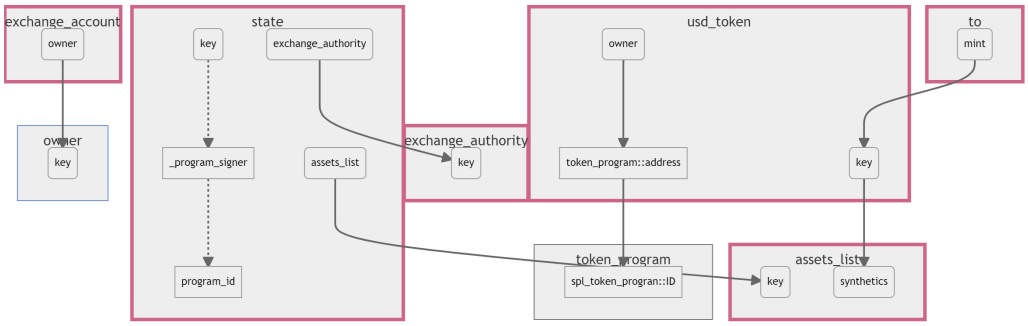


Figure 19: mint

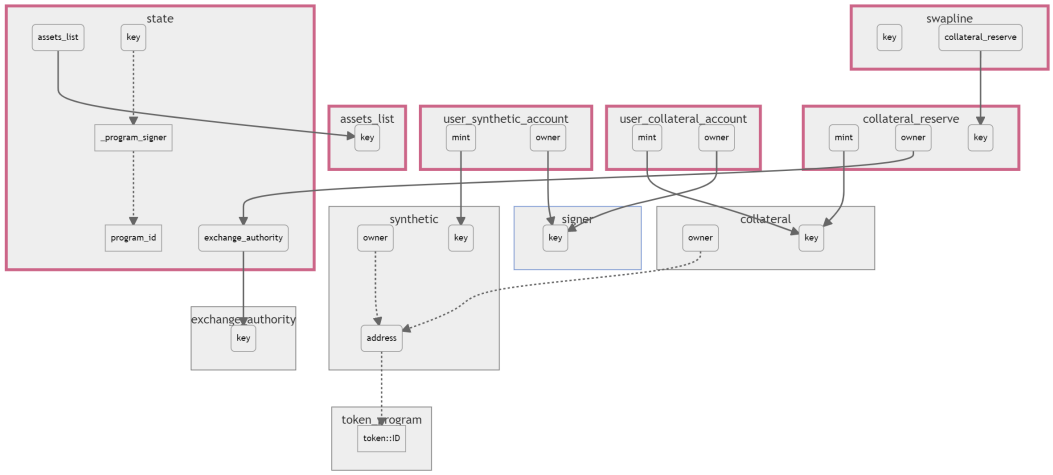


Figure 20: native_to_synthetic & synthetic_to_native

- Input accounts struct `RepayVault`
- Access control check on `state` halted and `vault` halted.

I2076

```
let synthetic = synthetics
    .iter_mut()
    .find(|x| {
        x.asset_address
            .eq(ctx.accounts.synthetic.to_account_info().key)
    })
    .unwrap();
```

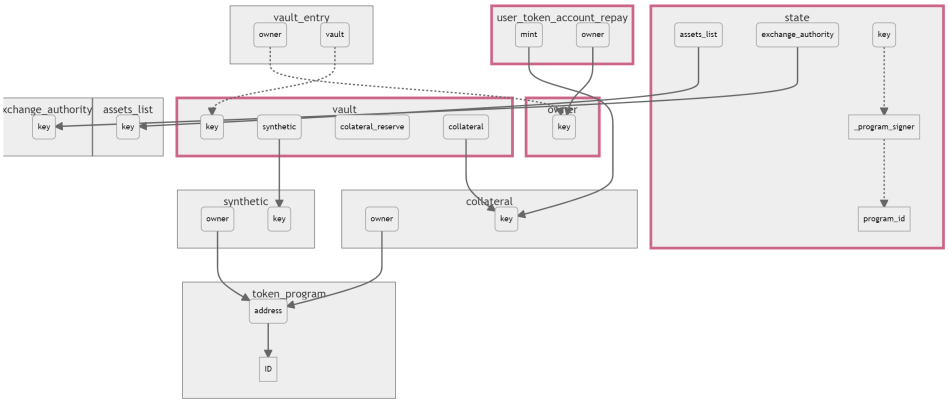


Figure 21: repay_draw

- Input accounts struct SetAdmin
- Access control check `state.admin == admin.key`

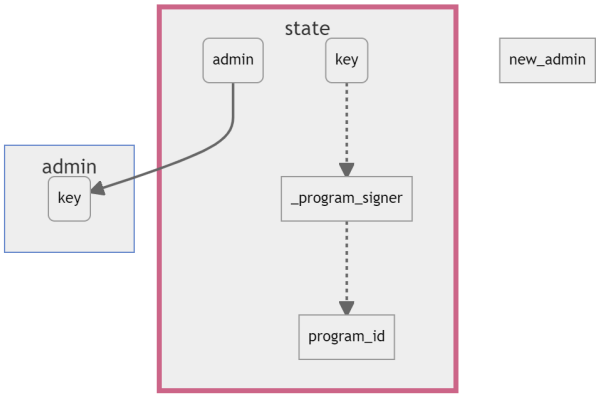


Figure 22: set_admin

- Input accounts struct is `SetAssetList`
- Before execution access_control tag verifies that ctx.accounts.state, ctx.accounts.admin.key==ctx.accounts.state.load().admin
- In this function `state.assets_list` is set to `Context<SetAssetList>.assets_list`

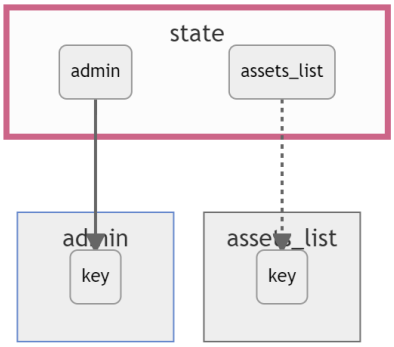


Figure 23: set_asssets_list

- Input accounts struct `SetCollateralRatio`.
- Access control check on `state.admin==admin.key`

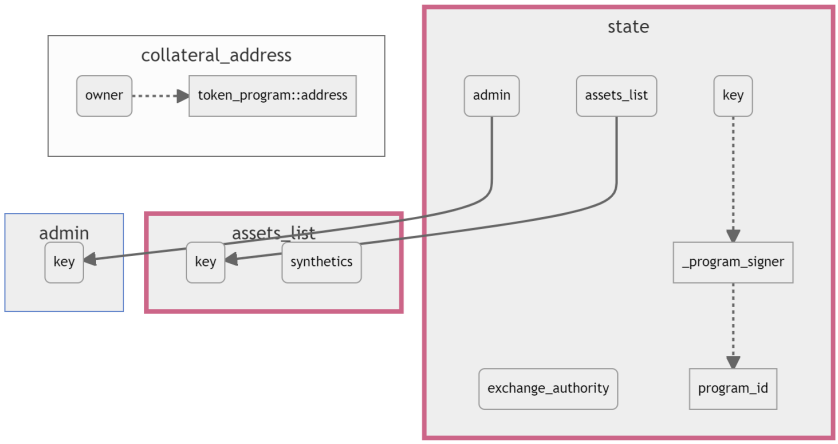


Figure 24: set_collateral_ratio

- Input accounts struct `SetHaltedSwapLine`
- Access control check `state.admin==admin.key`

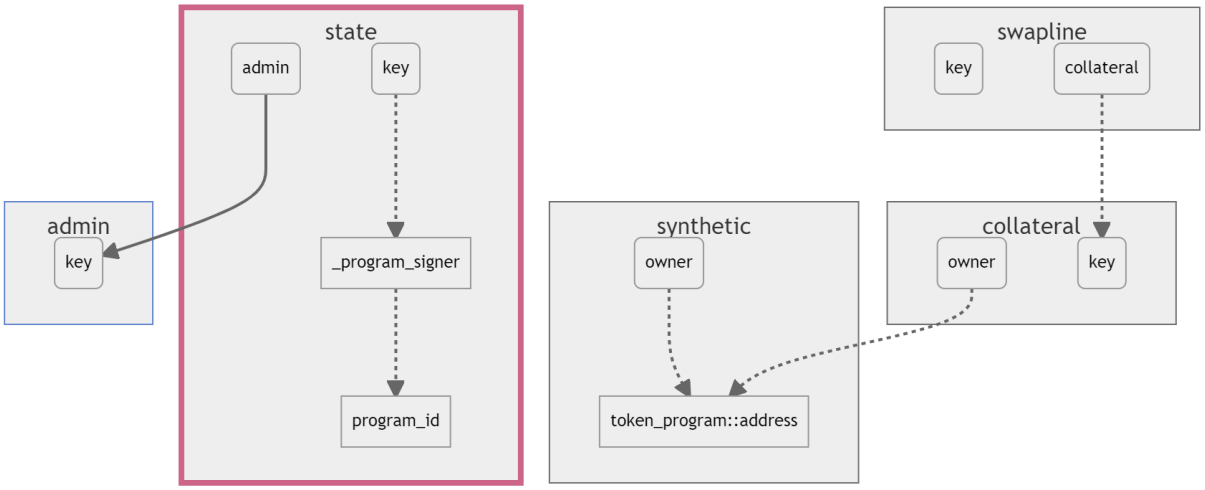


Figure 25: set_halted_swapline

- Access control on `state.admin==signer.key`.
- Input accounts struct is `SetMaxSupply`

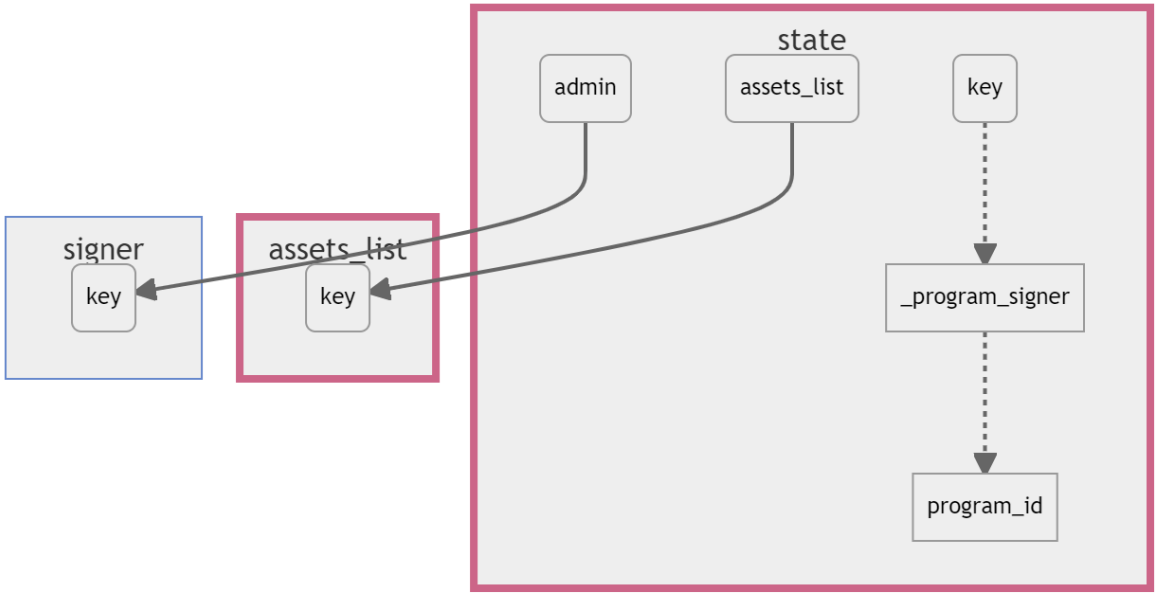


Figure 26: set_max_supply

- Access control on `state.admin==signer.key`.
- Input accounts struct is `SetPriceFeed`
- No explicit check on the price feed?

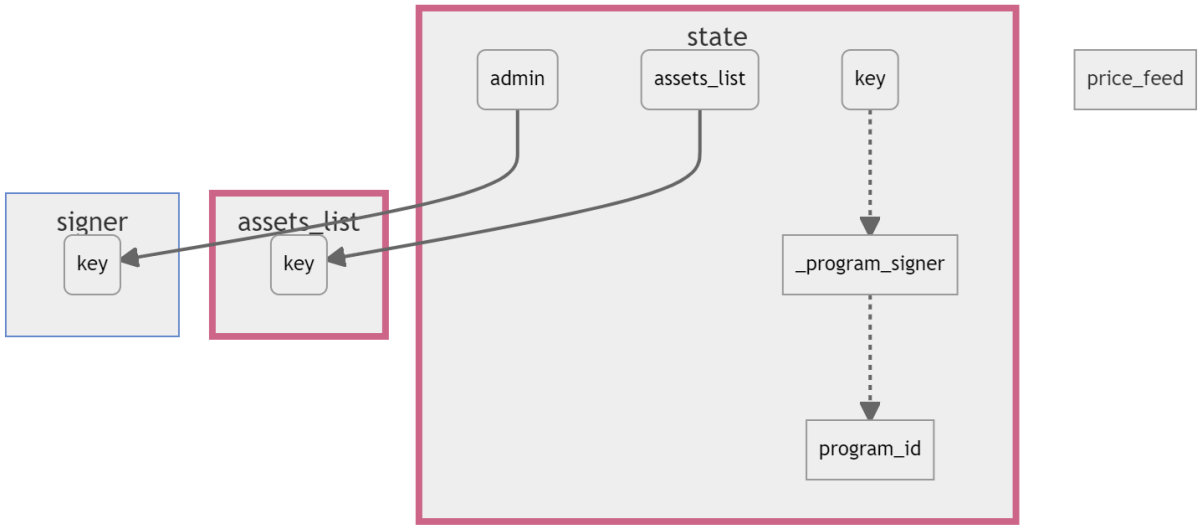


Figure 27: set_price_feed

- Input accounts struct `SetSettlementSlot`
- Access control check `state.admin==admin.key`

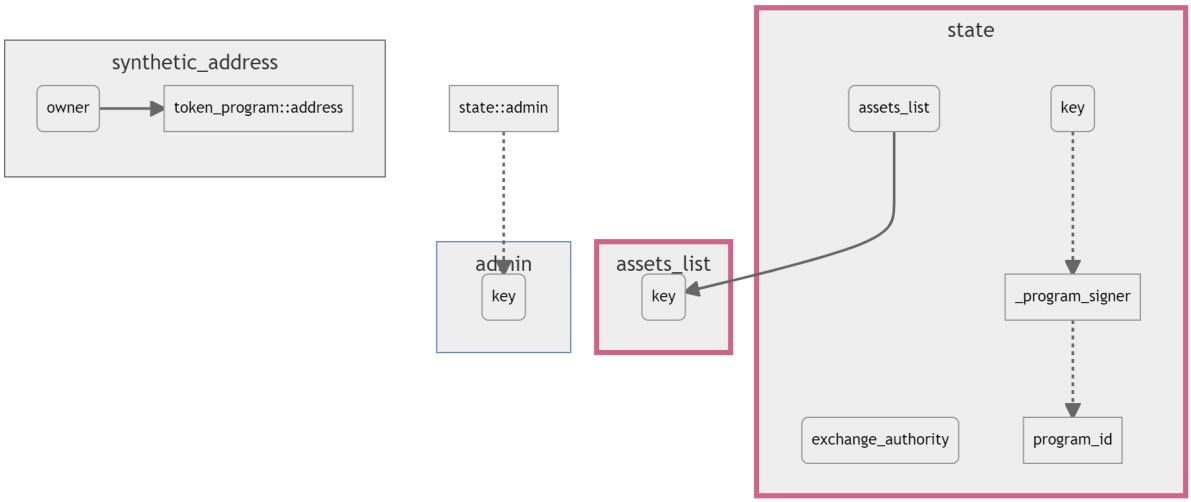


Figure 28: set_settlement_slot

- Input struct `SettleSynthetic`

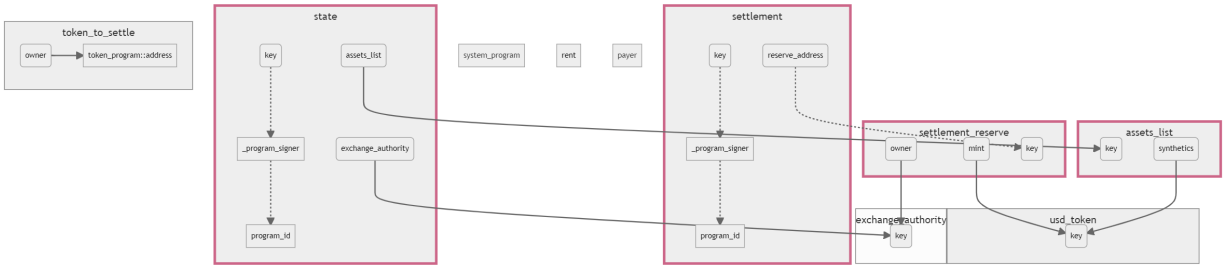


Figure 29: settle_synthetic

- Handle this explcitly (from l501)

```
        //Get indexes of both assets
        let synthetic_in_index = synthetics
            .iter()
            .position(|x| x.asset_address == *token_address_in)
            .unwrap();
        let synthetic_for_index = synthetics
            .iter()
            .position(|x| x.asset_address == *token_address_for)
            .unwrap();
```

- This simply returns an error. Is there any way to use the error to get the oracle to update? (l520)

```
        check_feed_update(
            assets,
            synthetics[synthetic_in_index].asset_index as usize,
            synthetics[synthetic_for_index].asset_index as usize,
            state.max_delay,
            slot,
        )
        .unwrap();
```

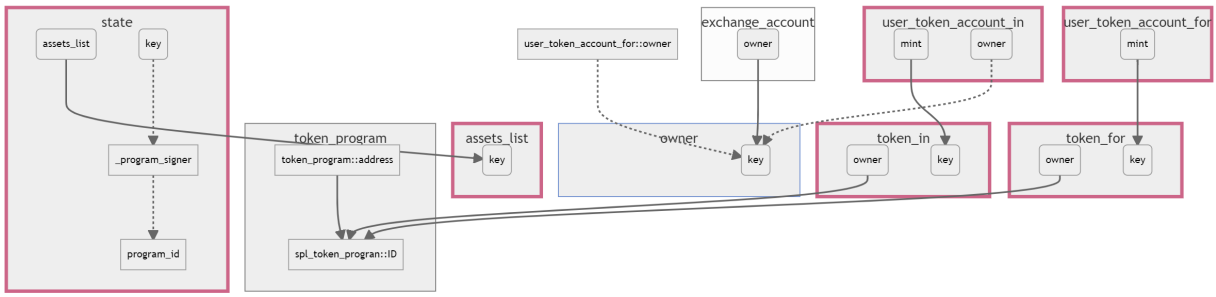


Figure 30: swap

- Input accounts struct `SwatSettledSynthetic`

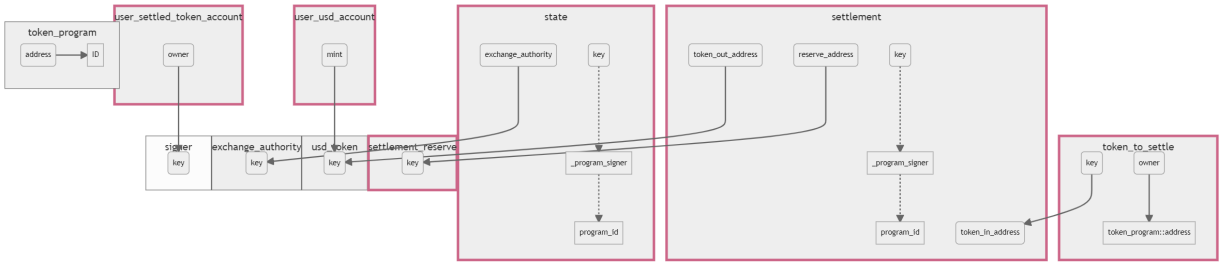


Figure 31: swap_settled_synthetic

- Access control check on `accounnts.state` to see if it is halted.

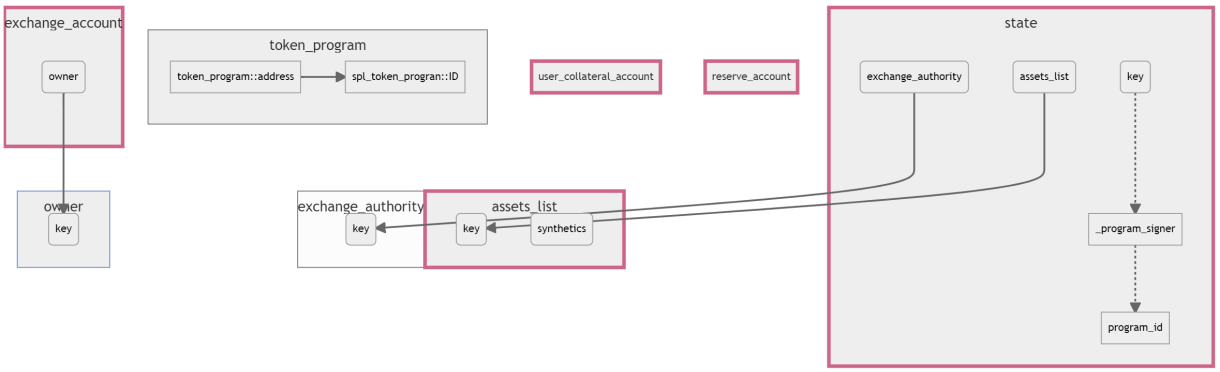


Figure 32: withdraw

- Input accounts struct is `WithdrawAccumulatedDebtInterest`
- Access control call on `state.admin == admin.key`

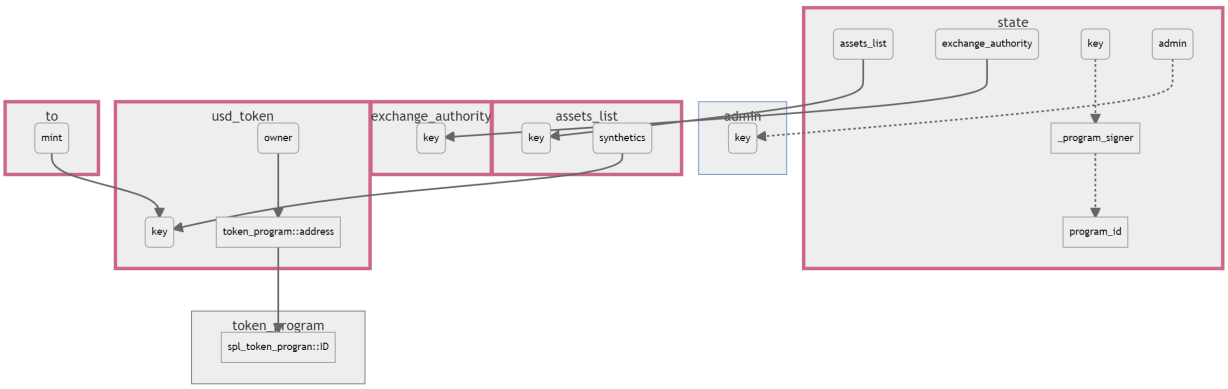


Figure 33: withdraw_accumulated_interest

line 1057

```
        let collateral = assets_list
            .collaterals
            .iter_mut()
            .find(|x| x.liquidation_fund.eq(liquidation_fund))
            .unwrap();
```

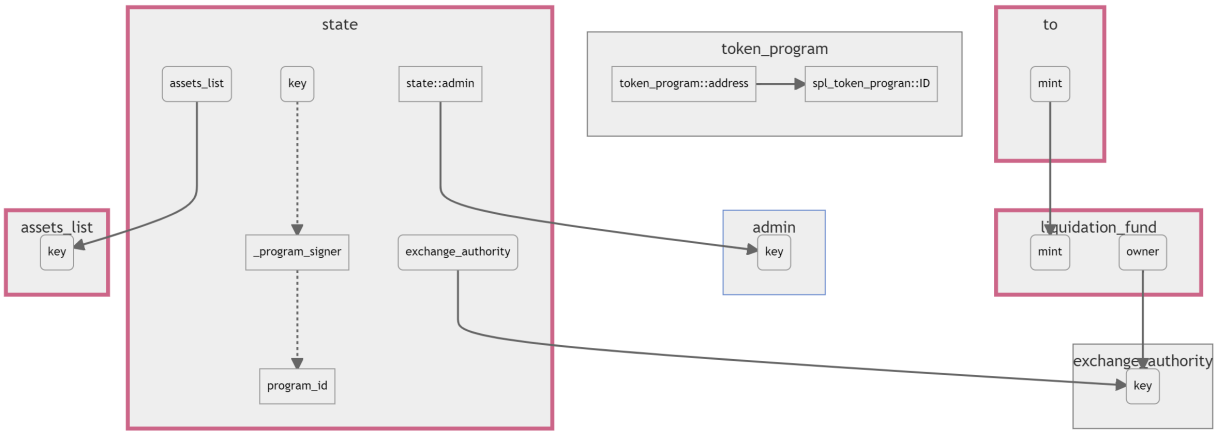


Figure 34: withdraw_liquidation_penalty

- Access control on `state.halted == True`

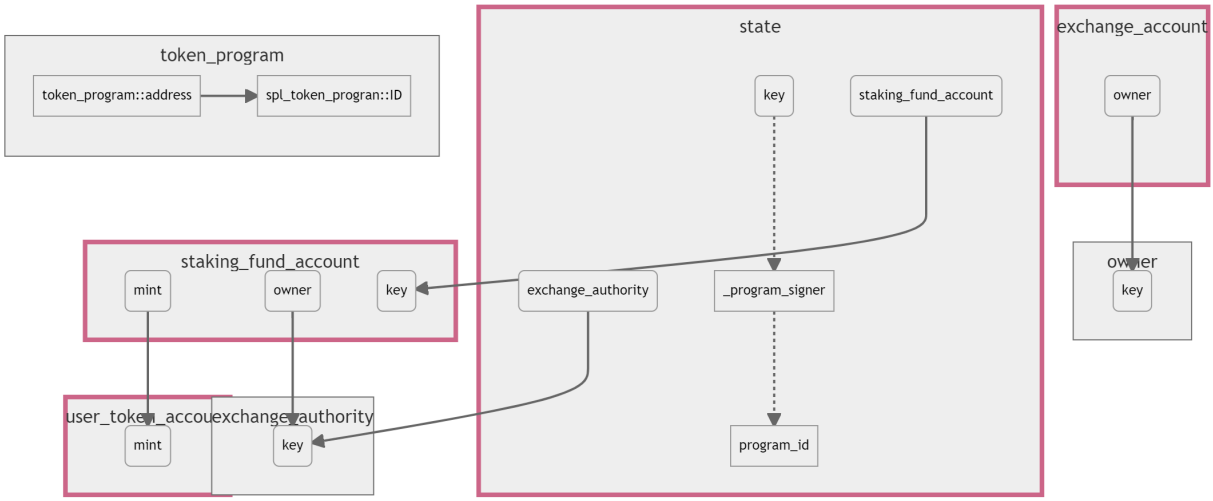


Figure 35: withdraw_rewards

- Input accounts struct is `AdminWithdraw`
- Access control check `state.admin == admin.key`

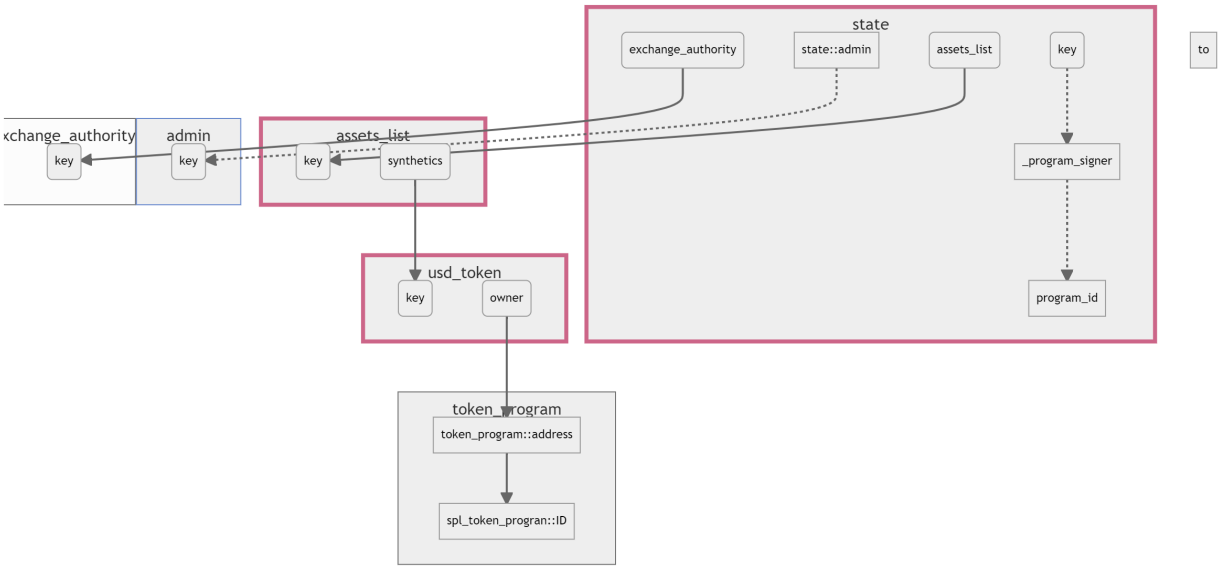


Figure 36: withdraw_swap_tax

- Input accounts struct `WithdrawSwapLineFee`
- Access control check `state.admin==admin.key`

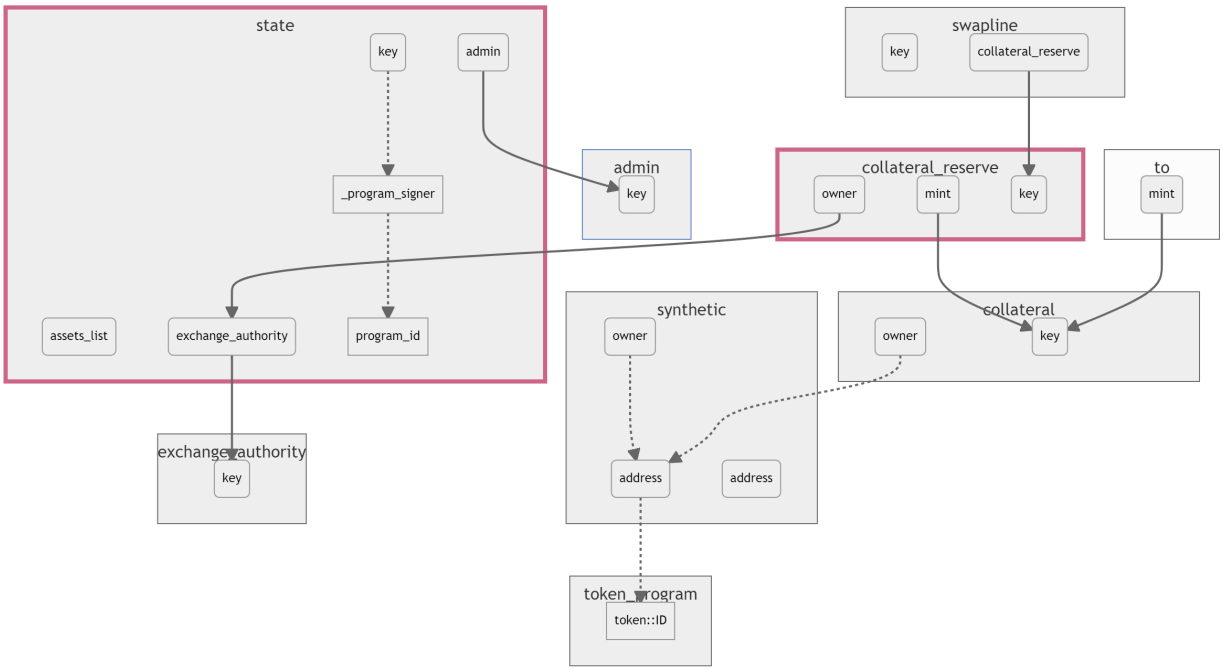


Figure 37: withdraw_swapline__fee

- Input accounts struct `WithdrawVault`
- Access control check on `state` halted and `vault` halted.

l2006

```
let synthetic_position = synthetics
    .iter_mut()
    .position(|x| {
        x.asset_address
            .eq(ctx.accounts.synthetic.to_account_info().key)
    })
    .unwrap();

let collateral_position = collaterals
    .iter()
    .position(|x| {
        x.collateral_address
            .eq(ctx.accounts.collateral.to_account_info().key)
    })
    .unwrap();
```

l2028

```
let vault_withdraw_limit = calculate_vault_withdraw_limit(
    collateral_asset,
    synthetic_asset,
    vault_entry.collateral_amount,
    vault_entry.synthetic_amount,
    vault.collateral_ratio,
)
.unwrap();
```

l2050

```
vault_entry.collateral_amount = vault_entry
    .collateral_amount
    .sub(amount_to_withdraw)
    .unwrap();
```

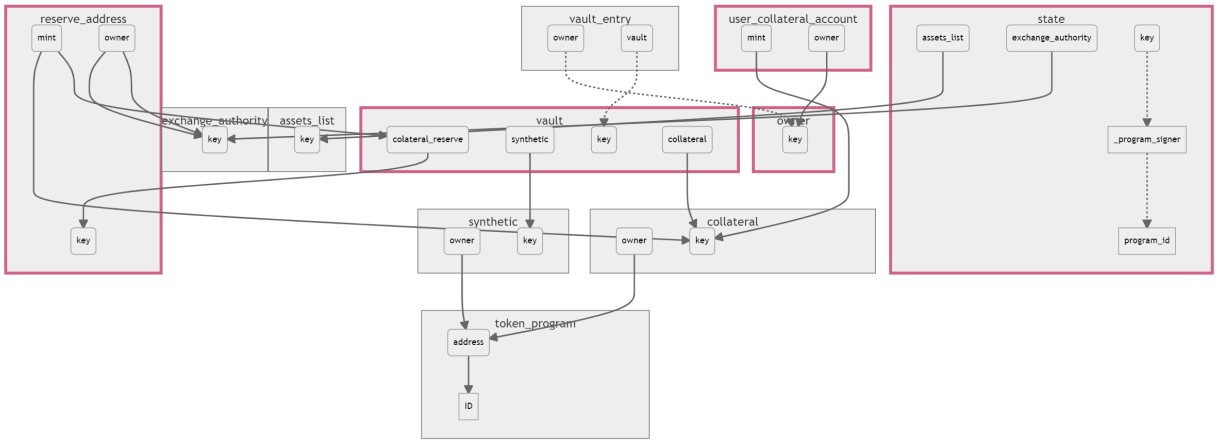


Figure 38: withdraw_vault

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.

Conclusion

Based on the account relationship graphs or reference graphs and the formal verification we can conclude that the code implements the documented functionality to the extent of the code reviewed.

Technical Findings

Extensive use of unwrap calls

Finding ID: KS-SYNTH-F-01

Severity: [Medium]

Status: [Remediated]

Description

The static code revealed 263 uses of the unwrap call in the code

Proof of Issue

Filename: programs/exchange/src/lib.rs (example)

Beginning Line Number: 118

```
116             match asset {
117                 Some(asset) => {
118                     let offset = (PRICE_SCALE as i32).checked_add(price_feed.expo).unwrap();
119                     if offset >= 0 {
120                         let scaled_price = price_feed
```

Severity and Impact Summary

The static code revealed 263 uses of the unwrap call in the code.

- Specific cases in system critical functions are discussed in individual issues.
- Use of the unwrap call in checked arithmetic should be checked. If it cannot be guaranteed that panics are avoided in all cases individual matching is recommended.

Unwraps can cause system crashes, which affect up-time and can cause memory dumps.

Recommendation

Check whether panic is guaranteed to be avoided. If not manage the cases ahould be managed manually.

References

N/A

Add constraints to InitializeAssetsList

Finding ID: KS-SYNTH-F-02

Severity: [Medium]

Status: [Remediated]

Description

- Before execution access_control tag verifies that `ctx.accounts.state`, `ctx.accounts.admin.key==ctx.accounts.state.load().admin`.

- Input ctx struct is `InitializeAssetsList`.

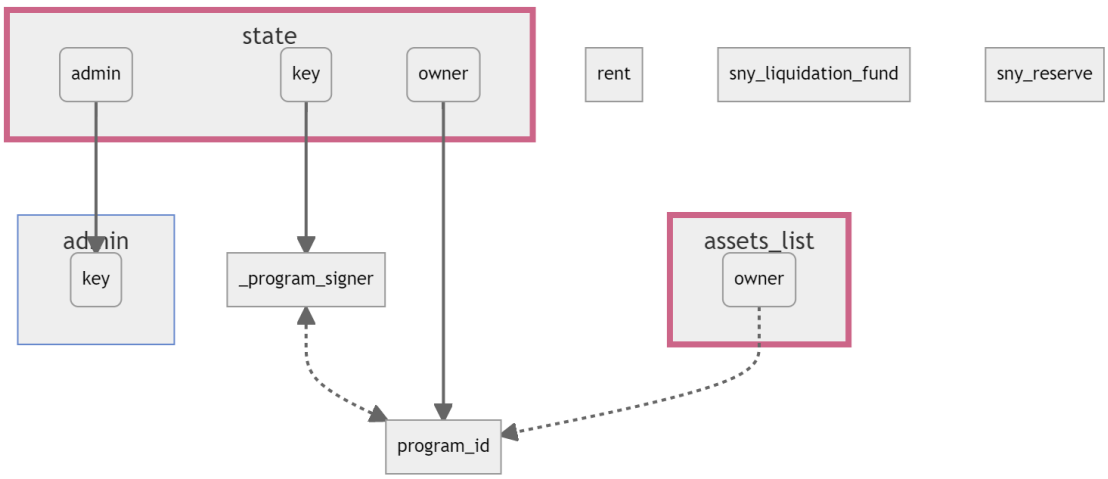
- `sny_reserve` and `sby_liquidation_fund` are used with no verification to initialize the `AssetList`. Unreferenced accounts can be replaced by others with malicious intent.

Proof of Issue

Filename: programs/exchange/src/context.rs

Beginning Line Number: 160

```
#[derive(Accounts)]
pub struct InitializeAssetsList<'info> {
    #[account(mut, seeds = [b"statev1".as_ref()],bump = state.load()?.bump)]
    pub state: Loader<'info, State>,
    #[account(zero)]
    pub assets_list: Loader<'info, AssetsList>,
    #[account(signer)]
    pub admin: AccountInfo<'info>,
    pub sny_reserve: AccountInfo<'info>,
    pub sny_liquidation_fund: AccountInfo<'info>,
    pub rent: Sysvar<'info, Rent>,
}
```



Severity and Impact Summary

Unreferenced accounts can potentially be supplanted my malicious actors.

Recommendation

While for many Solana instructions the SPL manages these issues, explicit checks are recommendable to ensure security and avoid possible new issues if there are changes to the SPL.

References

N/A

Use of unwrap is system critical functions

Finding ID: KS-SYNTH-F-03

Severity: [Medium]

Status: [Remediated]

Description

The use of unwraps in key functions, generally when searching for a given asset.

Proof of Issue

Filename: programs/exchange/src/lib.rs

Beginning Line Number: 1925 (these exists also at lines 1573, 1581, 1791, 1844, 256, 1880, 2153, 1657, 1722, 2076, 501, 1057, 2006, 2028)

```
let synthetic_position = synthetics
    .iter_mut()
    .position(|x| {
        x.asset_address
            .eq(ctx.accounts.synthetic.to_account_info().key)
    })
    .unwrap();

let collateral_position = collaterals
    .iter()
    .position(|x| {
        x.collateral_address
            .eq(ctx.accounts.collateral.to_account_info().key)
    })
    .unwrap();
```

Proof of Issue

Filename: programs/exchange/src/lib.rs

Beginning Line Number: 520

```
 check_feed_update(
      assets,
      synthetics[synthetic_in_index].asset_index as usize,
      synthetics[synthetic_for_index].asset_index as usize,
      state.max_delay,
      slot,
  )
  .unwrap();
```

This specific code may make it possible to use the error to trigger an oracle update..

Severity and Impact Summary

The unwraps should be managed manually and return success or appropriate errors. Unwrap calls can cause system panic, which can crash the program, affect uptime and dump critical information on memory.

Recommendation

Verify whether the unwrap calls can lead to error. If it is not certain that there is no way these calls can lead to error and hence panic, the cases should be handled manually with appropriate pattern matching.

References

N/A

Low test coverage

Finding ID: KS-SYNTH-F-04

Severity: [Low]

Status: [Remediated]

Description

Test report coverage reports 49% of lines (2052/4215), 17% (165/956) of functions covered by tests in `exchange`. ·

Test report coverage reports < 1% of lines (1/107), < 1% (1/150) of functions covered by tests in `pyth`.

Proof of Issue

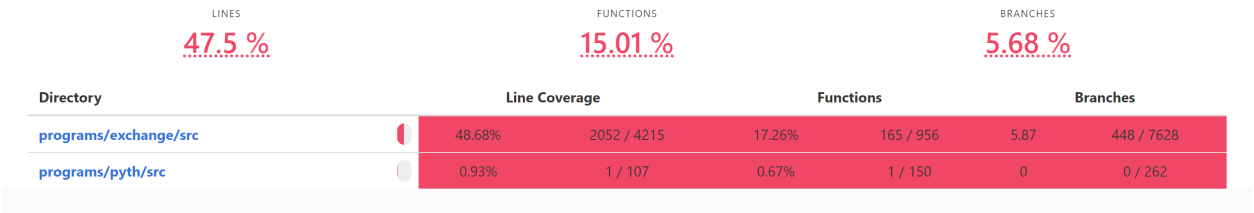


Figure 39: Test coverage

Severity and Impact Summary

Low test coverage can lead to problems in the maintenance of the code.

Recommendation

It is recommended that the team considers including more unit tests in the project to test remaining parts of code. This can aid maintenance and functionality checking.

References

N/A

Outdated packages and yanked references

Finding ID: KS-SYNTH-F-05

Severity: [**Low**]

Status: [**Remediated**]

Description

There are outdated and yanked depemdemcies in the codebase

Proof of Issue

References to outdated packages

```
exchange
================
Name       Project  Compat  Latest  Kind    Platform
----       -------  ------  ------  ----    --------
spl-token  3.1.1    ---     3.2.0   Normal  ---

pyth
================
Name      Project  Compat  Latest  Kind    Platform
----      -------  ------  ------  ----    --------
bytemuck  1.7.0    ---     1.7.2   Normal  ---
```

References to yanked crates

```
Crate:         bytemuck
Version:       1.7.0
Warning:       yanked
Dependency tree:
bytemuck 1.7.0
├── serum_dex 0.4.0
│   └── anchor-spl 0.13.2
│       ├── pyth 0.1.0
│       │   └── exchange 0.1.0
│       └── exchange 0.1.0
├── pyth 0.1.0
└── anchor-lang 0.13.2
    ├── pyth 0.1.0
    ├── exchange 0.1.0
    └── anchor-spl 0.13.2

Crate:         pyth
Version:       0.1.0
Warning:       yanked
Dependency tree:
pyth 0.1.0
└── exchange 0.1.0

warning: 2 allowed warnings found
```

Severity and Impact Summary

By referencing and creating dependencies on either outdated or yanked crates could introduce severe security issue as the reason for the state is unknown.

Recommendation

In the maintenance process of the code there should be a check to keep all dependencies up to date.

References

https://doc.rust-lang.org/cargo/reference/publishing.html#cargo-yank

Unreferenced accounts in account structs

Finding ID: KS-SYNTH-F-06

Severity: [Informational]

Status: [Remediated]

Description

There is Unreferenced, unchecked accounts in the structures that could cause issues.

Proof of Issue

The following diagrams describe these finidings in several account structs.

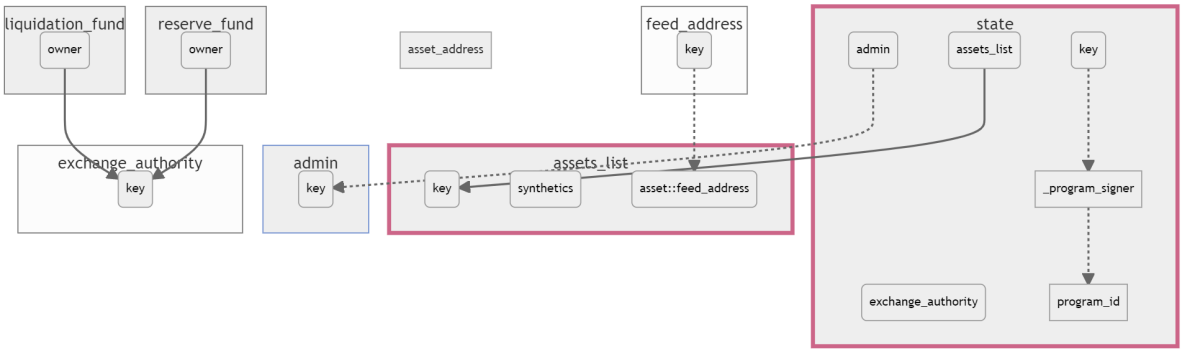


Figure 40: AddCollateral

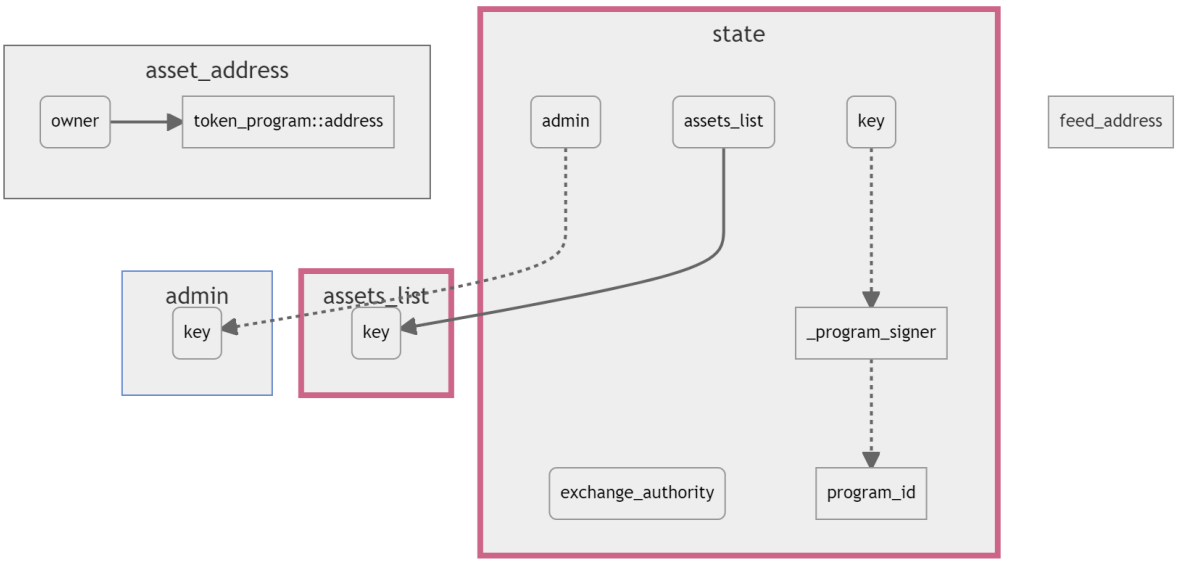


Figure 41: AddSynthetic

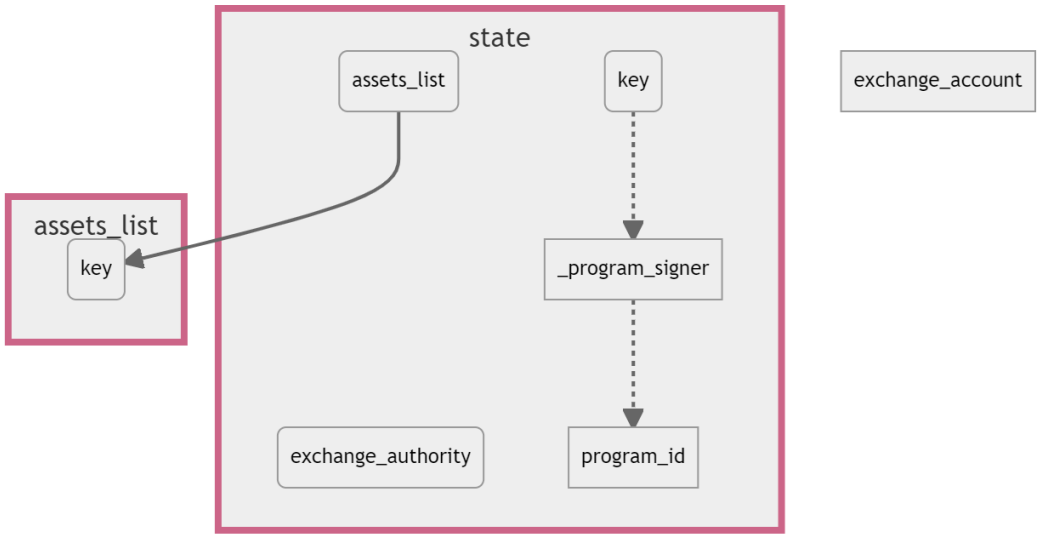


Figure 42: CheckAccountCollateralization

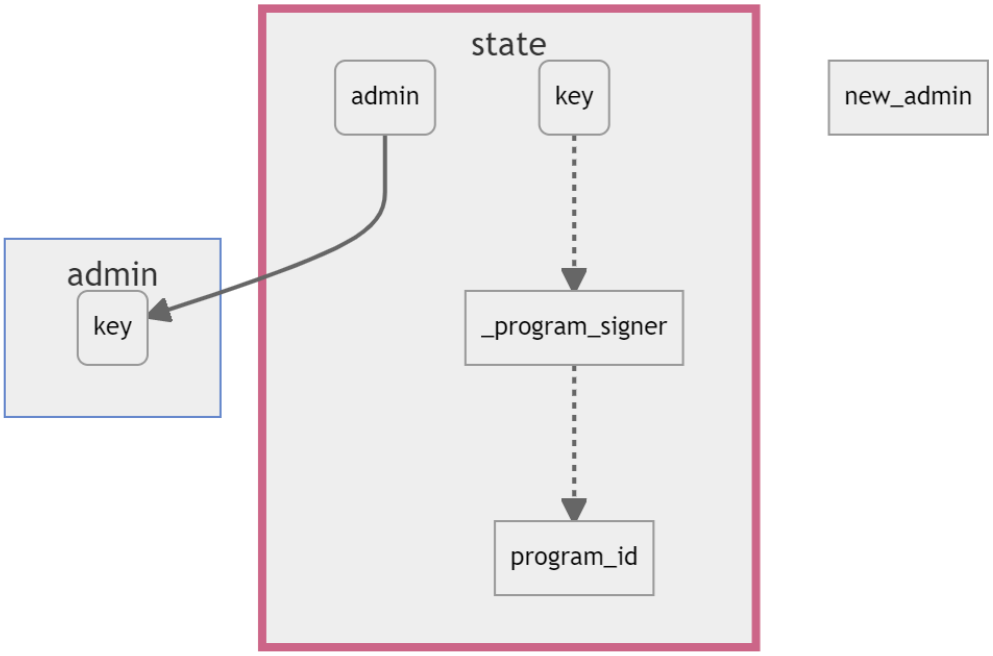


Figure 43: SetAdmin

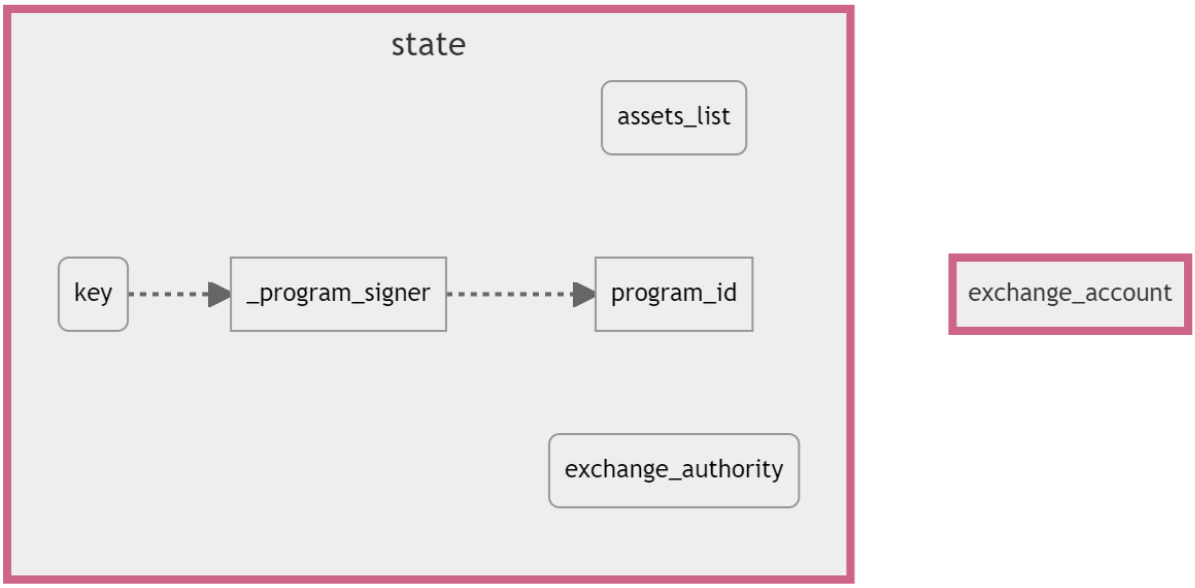


Figure 44: ClaimRewards

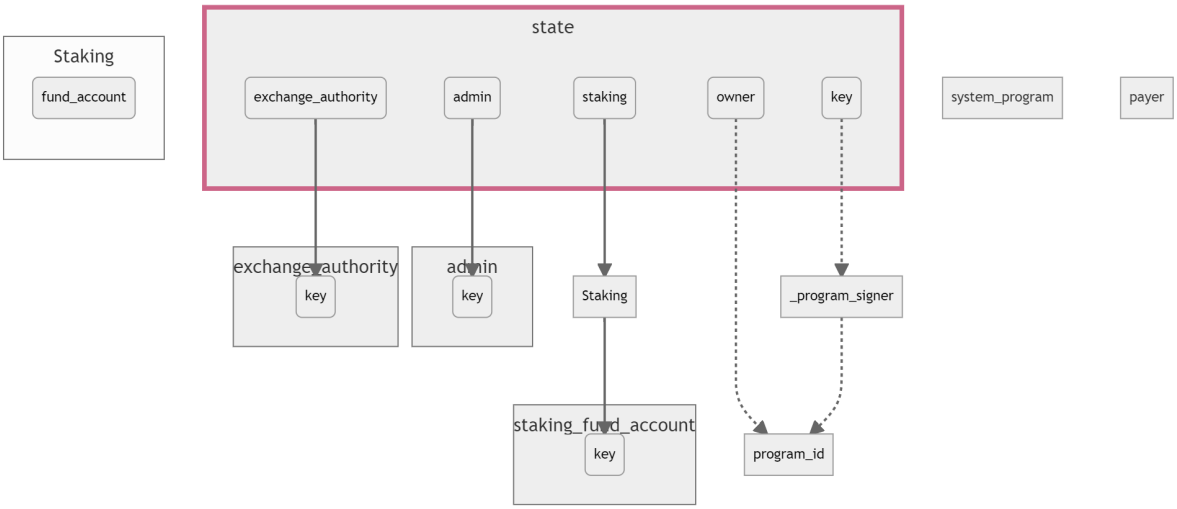


Figure 45: Init

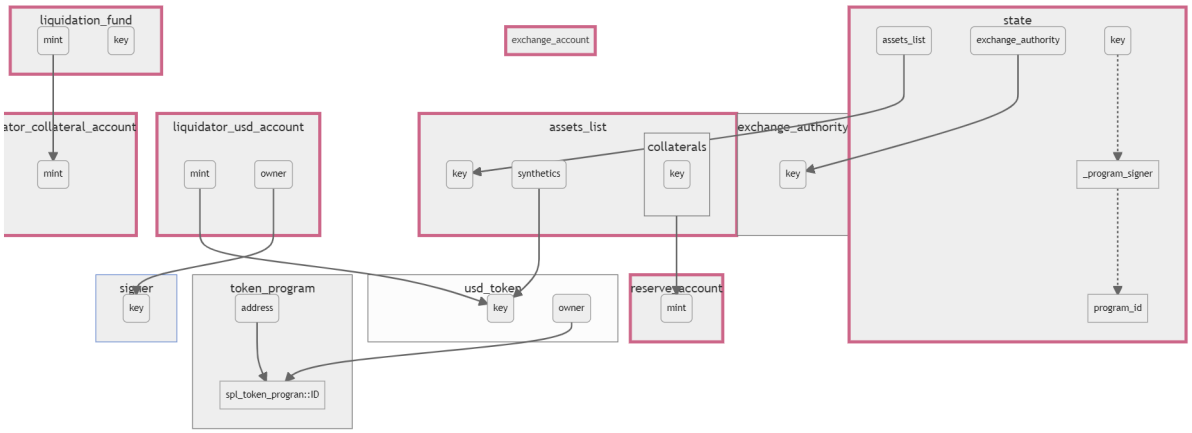


Figure 46: Liquidate

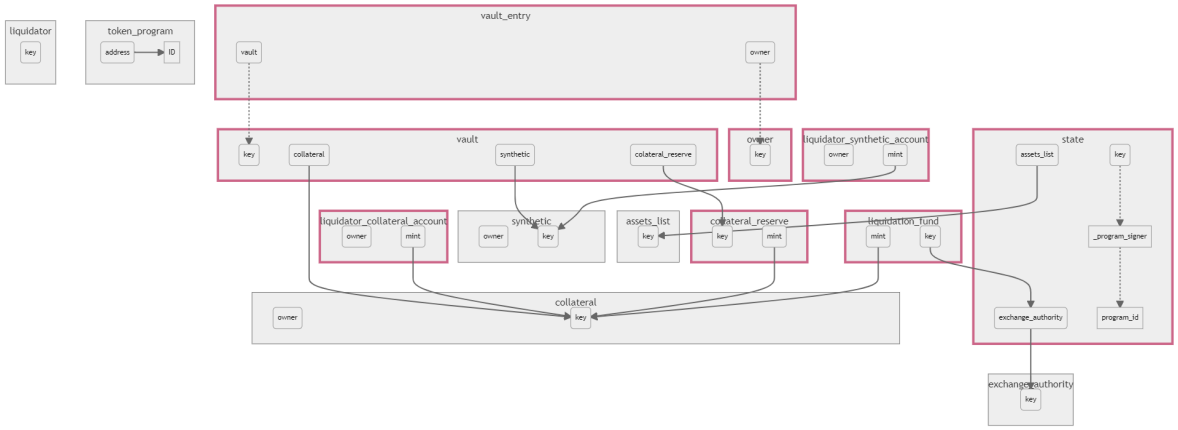


Figure 47: LiquidateVault

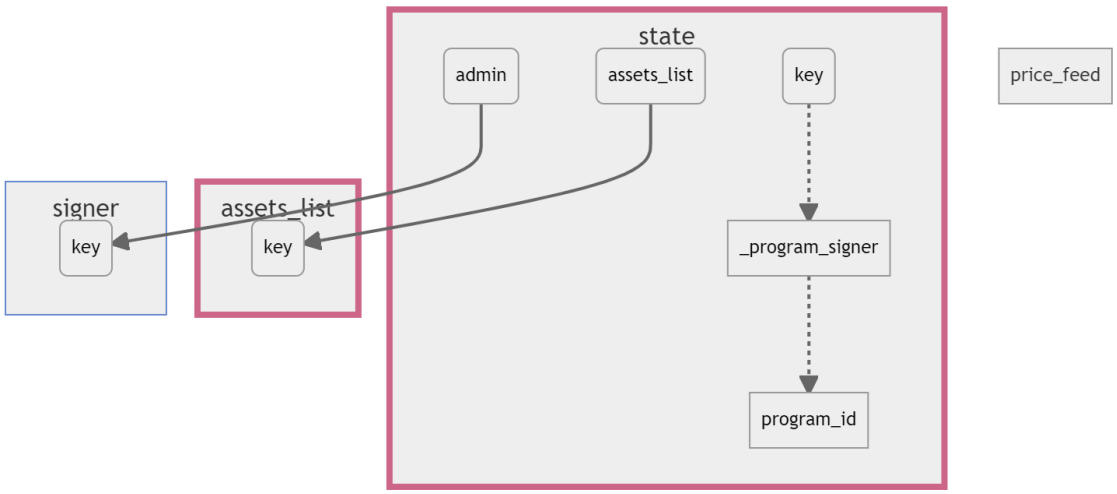


Figure 48: SetPriceFeed

Severity and Impact Summary

Unreferenced, unchecked accounts can cause errors or be replaced with malicious intent. It is good practice to include checks to avoid these situations.

Recommendation

Introduce checks so that the code doesn't rely on the security provided bu the blockchain or SPL. If there would be any change introduced there your code may inherit security issues.

References

N/A

METHODOLOGY

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

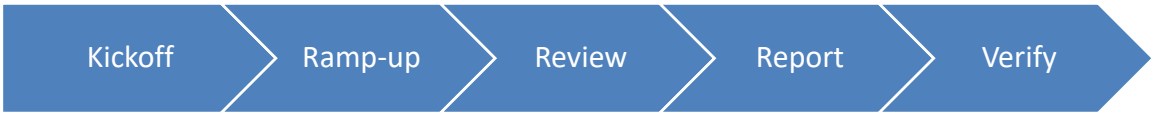


Figure 49: Methodology Flow

Kickoff

The project is kicked all of 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 a majority 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 as a whole. 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 probablillty 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 can not 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.

KUDELSKI SECURITY CONTACTS

NAME	POSITION	CONTACT INFORMATION
Scott Carlson	**Head of Blockchain Center of Excellence**	**Scottj.carlson@kudelskisecurity.com**