

Secure Code Assessment of the pSTAKE Solana Wrapper Contract

Findings and Recommendations Report Presented to:

pSTAKE Technologies PTE LTD

May 10, 2022

Version: 2.0

Presented by:

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

STRICTLY CONFIDENTIAL

TABLE OF CONTENTS

TABLE OF CONTENTS.....	2
LIST OF FIGURES	3
LIST OF TABLES	3
EXECUTIVE SUMMARY	4
Overview	4
Key Findings.....	4
Scope and Rules of Engagement	5
TECHNICAL ANALYSIS & FINDINGS	7
Findings.....	8
Technical Analysis	9
Authorization.....	9
Conclusion.....	9
Technical Findings	10
General Observations.....	10
Ownership of pool fee account (Authority)	11
Call restrictions with non-idiomatic use of anchor	12
Conflicting statements in documentation.....	13
Inconsistent account checks.....	14
Misleading comment.....	15
Missing doc comment for unchecked accounts.....	16
Redundant assignment of stake deposit authority	17
Use of 11 functions all named process	19
update_fee uses 8 params	20
Relationship Graphs.....	21
Privilege: Manager	23
Initialize.....	23
Update fee	24
Privilege: Staker	25
Add validator.....	25
Remove validator.....	26
Increase stake	27
Decrease stake.....	28
Privilege: User.....	29

Deposit	29
Deposit stake	30
Withdraw now	31
Withdraw later	32
Claim	33
METHODOLOGY	34
Kickoff	34
Ramp-up	34
Review	34
Code Safety	35
Technical Specification Matching	35
Reporting	35
Verify	36
Additional Note	36
The Classification of identified problems and vulnerabilities	36
Critical – vulnerability that will lead to loss of protected assets	36
High - A vulnerability that can lead to loss of protected assets	36
Medium - a vulnerability that hampers the uptime of the system or can lead to other problems	37
Low - Problems that have a security impact but does not directly impact the protected assets	37
Informational	37
Tools	38
RustSec.org	38
KUDELSKI SECURITY CONTACTS	39

LIST OF FIGURES

Figure 1: Findings by Severity	7
Figure 2: Methodology Flow	34

LIST OF TABLES

Table 1: Scope	6
Table 2: Findings Overview	8
Table 3: Legend for relationship graphs	22

EXECUTIVE SUMMARY

Overview

pSTAKE Technologies PTE LTD engaged Kudelski Security to perform a Secure Code Assessment of the pSTAKE Solana Wrapper Contract.

The basic idea behind pSTAKE Solana is to allow liquid SOL staking and unstaking. The purpose of the stake pool is to delegate user's SOL. In return, it will offer derivative tokens i.e., stkSOL that represents the user's stakedSOL. By this, users get the chance to gain extra yield by utilizing their liquid token (stkSOL).

The assessment was conducted remotely by the Kudelski Security Team. Testing took place on March 07 - March 18, 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

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

- KS-PSTAKE-01 – Ownership of pool fee account (Authority)
- KS-PSTAKE-02 – Call restrictions with non-idiomatic use of anchor
- KS-PSTAKE-03 – Conflicting statements in documentation
- KS-PSTAKE-04 – Inconsistent account checks
- KS-PSTAKE-05 – Misleading comment
- KS-PSTAKE-06 – Missing doc comment for unchecked accounts
- KS-PSTAKE-07 – Redundant assignment of stake deposit authority
- KS-PSTAKE-08 – Use of 11 functions all named process
- KS-PSTAKE-09 – update_fee uses 8 params

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 pSTAKE Secure Code Assessment. 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/persistenceOne/pSTAKE-> with the commit hash 4ead3f1990e9c12a60e77602e59f267ecc5c3b70. A re-review was performed on April 25, 2022, with commit hash 1885c96a347d029c5c9222e4f81d01bf11ed37cb.

Files included in the code review

```
pstake/
├── contract/
│   └── wrapper-contract/
│       ├── artifacts/
│       │   └── spl_stake_pool.so
│       ├── migrations/
│       │   └── deploy.ts
│       ├── programs/
│       │   └── wrapper-contract/
│       │       ├── src/
│       │       │   ├── instructions/
│       │       │   │   ├── add_validator.rs
│       │       │   │   ├── claim.rs
│       │       │   │   ├── decrease_stake.rs
│       │       │   │   ├── deposit.rs
│       │       │   │   ├── deposit_stake.rs
│       │       │   │   ├── increase_stake.rs
│       │       │   │   ├── initialize.rs
│       │       │   │   ├── remove_validator.rs
│       │       │   │   ├── update_fee.rs
│       │       │   │   ├── withdraw_later.rs
│       │       │   │   └── withdraw_now.rs
│       │       │   ├── state/
│       │       │   │   ├── contract_state.rs
│       │       │   │   └── validators_score_list.rs
│       │       │   ├── utils/
│       │       │   │   ├── pda_seeds.rs
│       │       │   │   └── program_accounts.rs
│       │       │   ├── error.rs
│       │       │   ├── instructions.rs
│       │       │   ├── lib.rs
│       │       │   ├── state.rs
│       │       │   └── utils.rs
│       │       ├── Cargo.toml
│       │       └── Xargo.toml
│       └── scripts/
│           └── install_spl_stake_pool.sh
```

	└─ test.sh
	└─ target/
	└─ deploy/
	└─ wrapper_contract-keypair.json
	└─ tests/
	└─ constants.ts
	└─ pdas.ts
	└─ utils.ts
	└─ wrapper-contract.ts
	└─ Anchor.toml
	└─ Cargo.toml
	└─ Makefile
	└─ README.md
	└─ deploy.js
	└─ package.json
	└─ tsconfig.json
	└─ yarn.lock
└─ multisig/	└─ README.md
└─ off-chain/	└─ README.md
└─ README.md	
└─ pSTAKE-solana-doc.md	
└─ pStake-solana-img.jpeg	

Table 1: Scope

TECHNICAL ANALYSIS & FINDINGS

During the pSTAKE Secure Code Assessment, we discovered:

- 1 finding with CRITICAL severity rating.
- 8 findings with INFORMATIONAL severity rating.

The following chart displays the findings by severity.

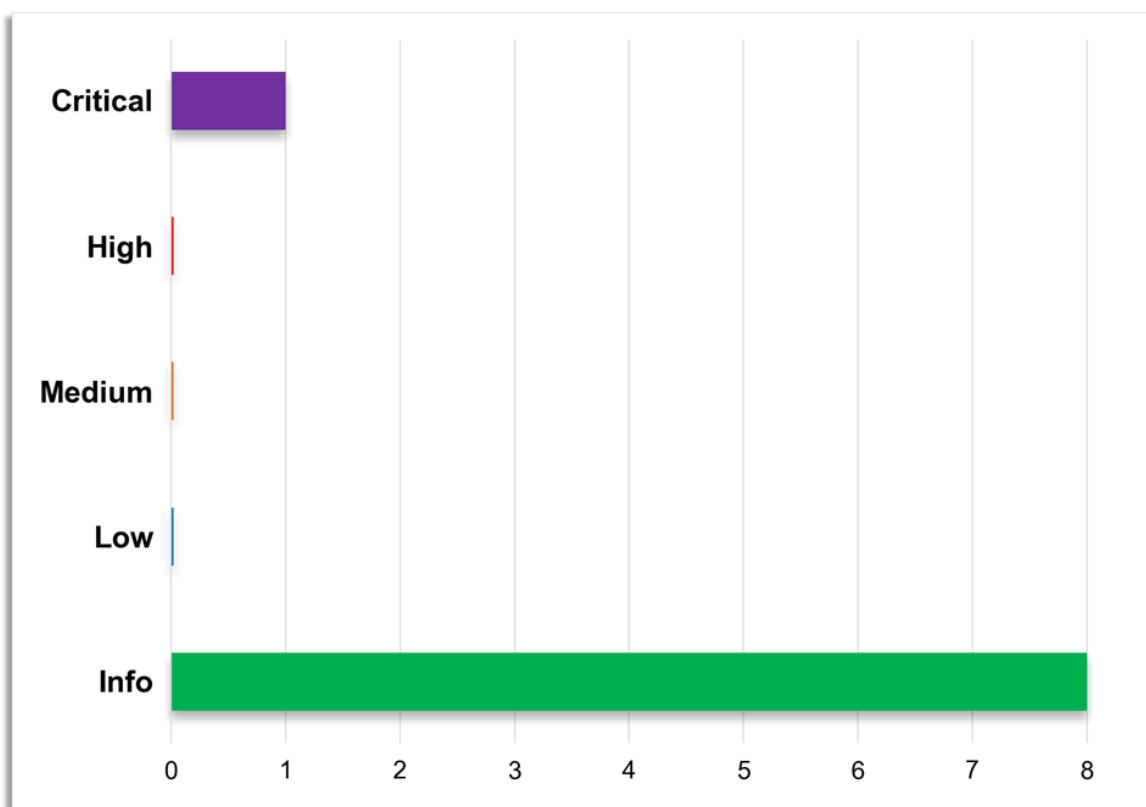


Figure 1: Findings by Severity

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-PSTAKE-01	Critical	Ownership of pool fee account (Authority)
KS-PSTAKE-02	Informational	Call restrictions with non-idiomatic use of anchor
KS-PSTAKE-03	Informational	Conflicting statements in documentation
KS-PSTAKE-04	Informational	Inconsistent account checks
KS-PSTAKE-05	Informational	Misleading comment
KS-PSTAKE-06	Informational	Missing doc comment for unchecked accounts
KS-PSTAKE-07	Informational	Redundant assignment of stake deposit authority
KS-PSTAKE-08	Informational	Use of 11 functions all named process
KS-PSTAKE-09	Informational	update_fee uses 8 params

Table 2: 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.

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 analysis of the relationship graphs. For details, see section Relationship Graphs starting on page 21.

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 the code assessment, it was noted that the code is well structured and concisely split into files. The rust code is well written and the use of checked arithmetic operations to protect from overflow/underflow operations shows commitment to writing secure programs. The use of *cargo fmt* to format code, as well as the use of other rust tools to verify/audit code such as *cargo clippy*, *audit* and *geiger* were appreciated and show a high level of expertise with the rust programming language.

The code documentation is decent, however doc comments required by anchor were missing. The use of the anchor framework with its inbuilt account verification functionality is a great foundation for this Solana project. The style in which anchor was used however is closer to a conventional Solana program. It is advisable to strongly rely on anchors inbuilt validation and access restriction macros.

Anchor provides its own build command with `anchor build` which will also generate an *IDL* file for client generation. This build command fails due to the aforementioned absence of doc comments required when using *UncheckedAccount*.

Ownership of pool fee account (Authority)

Finding ID: KS-PSTAKE-01

Severity: **Critical**

Status: **Remediated**

Description

The account.data.owner (authority) of the pool fee account is set to the token program. This implies that the fees collected in this account can only ever be accessed by the Solana team that wrote the token program.

The idea is that this account collects fees for the manager who manages the stake pool, adds/removes validators. Users who stake, pay some configurable fee into that account. However, as stated above, the manager will not be able to transfer funds out of the fee account.

It appears, judging by the comment in the code below, that this issue was known to the developers.

Proof of Issue

File name: initialize.rs

Line number: 230

```
spl_token::instruction::initialize_account(  
    self.token_program.key,  
    self.pool_fee.key,  
    self.pool_mint.key,  
    self.token_program.key, // TODO (bug): shouldn't this contract be the owner?  
)?,
```

Severity and Impact Summary

While the account itself (account.owner) should be owned by the token program, the account.data.owner (token account authority) should be given to the account that's authorized to transfer tokens from that account. As implemented, the pool_fee token account will not be accessible to anyone but the developer of the Solana token program. This is a critical issue, as all fees collected by the stake pool will not be accessible to the manager.

Recommendation

Assign an account that is authorized to transfer tokens.

Remediation

File name: initialize.rs

Line number: 258

```
&spl_token::instruction::initialize_account(  
    self.token_program.key,  
    self.pool_fee.key,  
    self.pool_mint.key,  
    self.manager.key, // only manager will be able to use the fee account  
)?,
```

Call restrictions with non-idiomatic use of anchor

Finding ID: KS-PSTAKE-02

Severity: **Informational**

Status: **Remediated**

Description

This code is meant to restrict access based on custom checks. Anchor provides functionality based on macros to facilitate custom checks. It is advisable to implement restrictions in a style that is coherent with anchor.

Proof of Issue

File name: lib.rs

Line number: 20

```
pub fn initialize(ctx: Context<Initialize>, params: InitializeParams) ->
ProgramResult {
    check_context(&ctx)?;
    ...
}

/// Perform some common checks on the context. This should be done by every
instruction.
fn check_context<T>(ctx: &Context<T>) -> ProgramResult {
    ...
}
```

Severity and Impact Summary

It would be advantageous to implement checks in a way that is coherent with the anchor framework so that it is more easily understood by other developers.

Recommendation

Use anchor provided macros such as:

```
#[access_control]
```

References

https://docs.rs/anchor-attribute-access-control/latest/anchor_attribute_access_control/attr.access_control.html

Conflicting statements in documentation

Finding ID: KS-PSTAKE-03

Severity: **Informational**

Status: **Remediated**

Description

It is stated in pSTAKE-solana-doc.md that:

Deposits do not change the exchange rate since we add new stkSOL 1:1 to the amount deposited in SOL, so it keeps the rate constant.

And later:

Mint SPL(stkSOL) tokens to the user's SPL token account (stk user lamport account) based on the exchange rate.

Proof of Issue

File name: pSTAKE-solana-doc.md

Line number: 31

Severity and Impact Summary

The documentation should stick to one formulation of how many tokens are issued.

Recommendation

State that tokens are issued based on the current exchange rate.

Inconsistent account checks

Finding ID: KS-PSTAKE-04

Severity: **Informational**

Status: **Remediated**

Description

Some of the account checks being done in macros are inconsistent across the program.

Proof of Issue

File name: deposit_stake.rs

Line number: 66

```
#[account(  
    mut,  
    seeds = [pda_seeds::POOL_FEE],  
    bump = contract_state.bump_seeds.pool_fee_bump,  
    owner = token::ID,  
)]  
pub pool_fee: UncheckedAccount<'info>,
```

File name: withdraw_later.rs

Line number: 72

```
#[account(  
    mut,  
    seeds = [pda_seeds::POOL_FEE],  
    bump = contract_state.bump_seeds.pool_fee_bump,  
)]  
pub pool_fee: UncheckedAccount<'info>,
```

Severity and Impact Summary

In deposit_stake the account ownership of the PDA is checked, yet in withdraw_later it is not. There are other examples such as the reserve_stake account, the pool_mint account, the stake_pool_state account.

Recommendation

It would be best to keep the checks consistent across the program.

Misleading comment

Finding ID: KS-PSTAKE-05

Severity: **Informational**

Status: **Remediated**

Description

In the withdraw function there is a comment for an account that says it will receive pool tokens. This comment has likely been copied from deposit.rs where it is accurate.

Proof of Issue

File name: withdraw_later.rs

Line number: 83

```
/// User account to receive pool tokens.
```

Severity and Impact Summary

The comment is misleading.

Recommendation

It would be best to have accurate comments.

Missing doc comment for unchecked accounts

Finding ID: KS-PSTAKE-06

Severity: **Informational**

Status: **Remediated**

Description

Anchor requires the use of UncheckedAccount to be annotated with a doc comment:

```
/// CHECK: Explanation of why this does not need to be checked  
pub some_account: UncheckedAccount<'info>
```

Proof of Issue

There are 62 cases in 11 files.

Severity and Impact Summary

It would be good practice to indicate the thinking behind why some accounts do not require any checks.
Also the use of `anchor build` fails since these doc comments are missing.

Recommendation

It would be good practice to add the missing doc comments to indicate why an account does not require checks.

References

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

Redundant assignment of stake deposit authority

Finding ID: KS-PSTAKE-07

Severity: **Informational**

Status: **Remediated**

Description

The stake deposit authority of the stake pool is assigned to the same account twice.

Proof of Issue

File name: initialize.rs

Line number: 242

```
invoke_signed(  
    &spl_stake_pool::instruction::initialize(  
        ...  
        Some(self.funding_authority.key()),  
        ...  
    ),  
    ...  
)?;  
  
invoke(  
    &spl_stake_pool::instruction::set_funding_authority(  
        ...  
        Some(self.funding_authority.key),  
        spl_stake_pool::instruction::FundingType::StakeDeposit,  
    ),  
    ...  
)?;
```

Severity and Impact Summary

The stake pool program assigns both the stake deposit authority and sol deposit authority if the deposit_authority is given (see the reference section for the relevant code). The code above will redundantly assign the stake deposit authority. While this does not impact the functionality of the program, it does run unnecessary instructions.

Recommendation

It could be advantageous to avoid unnecessary instructions for code compactness and possible optimization even though Solana does not currently use an Ethereum style gas cost calculation.

References

<https://solana.wiki/docs/solidity-guide/transactions/>

<https://github.com/solana-labs/solana-program-library/blob/master/stake-pool/program/src/processor.rs#L649>

```
let (stake_deposit_authority, sol_deposit_authority) = match  
next_account_info(account_info_iter) {  
    Ok(deposit_authority_info) => (
```

```
        *deposit_authority_info.key,  
        Some(*deposit_authority_info.key),  
    ),  
    ...  
};  
...  
stake_pool.stake_deposit_authority = stake_deposit_authority;  
...  
stake_pool.sol_deposit_authority = sol_deposit_authority;
```

Use of 11 functions all named process

Finding ID: KS-PSTAKE-08

Severity: **Informational**

Status: **Remediated**

Description

There are 11 different functions all called *process*.

Proof of Issue

File name: instructions/*.rs

Severity and Impact Summary

There are 26 functions of which 11 are called *process*, yet there is no code related need for this.

Recommendation

It would be better practice to name these functions based on what they do, thus increasing code readability/clarity.

update_fee uses 8 params

Finding ID: KS-PSTAKE-09

Severity: **Informational**

Status: **Remediated**

Description

This function uses 8 parameters. As otherwise done, it would be good to use a struct for the parameters.

Proof of Issue

File name: lib.rs

Line number: 70

```
pub fn update_fee(  
    ctx: Context<UpdateFee>,  
    sol_referral: Option<u8>,  
    stake_referral: Option<u8>,  
    epoch: Option<Fee>,  
    sol_deposit: Option<Fee>,  
    stake_deposit: Option<Fee>,  
    sol_withdrawal: Option<Fee>,  
    stake_withdrawal: Option<Fee>,  
) -> ProgramResult { .. }
```

Severity and Impact Summary

It would be best to follow general rust programming guidelines to keep the code more readable.

Recommendation

Combine the parameters into a struct.

References

https://rust-lang.github.io/rust-clippy/master/index.html#too_many_arguments

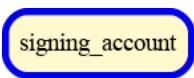
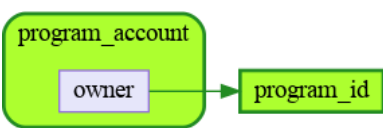
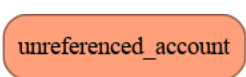
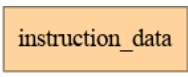
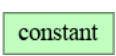
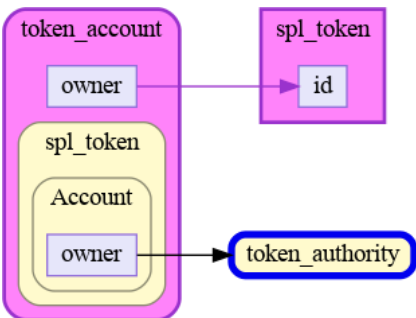
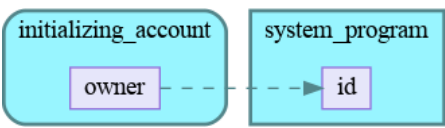
Relationship Graphs

A relationship graph shows relational requirements to the input of an instruction. Notice, that it does not show outcome of the instruction.

Relationship graphs are used to analyze insufficient authorization and unchecked account relations. Insufficient authorization allows unintended access. Unchecked account relations may allow account and data injection resulting in unintended behavior and access.

Various styles are used to highlight special properties. Accounts are shown as boxes with round corners. An account box may contain smaller boxes indicating relevant account data.

The following table shows the basic styles for the relationship graphs.

	Round boxes with thick blue borders indicate accounts required to sign the transaction.
	Green round boxes highlight accounts required to be owned by the program itself.
	Red round boxes indicate accounts where ownership is not validated. Further analysis should be made to ensure this does not allow account injection attacks.
	Orange boxes indicate instruction data. As instruction data does not origin from the blockchain, it may open for data injection attacks. Caution must be taken if used for account validation.
	Green boxes indicate constants. It may refer to either be hardcoded values or library code.
	Inner boxes are used to indicate data structures to ease readability. Additional colors may also be used to highlight account ownership from different programs.
	Dashed lines indicate implicitly required relations. For example, relations required by another program during a cross program invocation. This is emphasized as the program cannot guarantee the behavior of external programs.

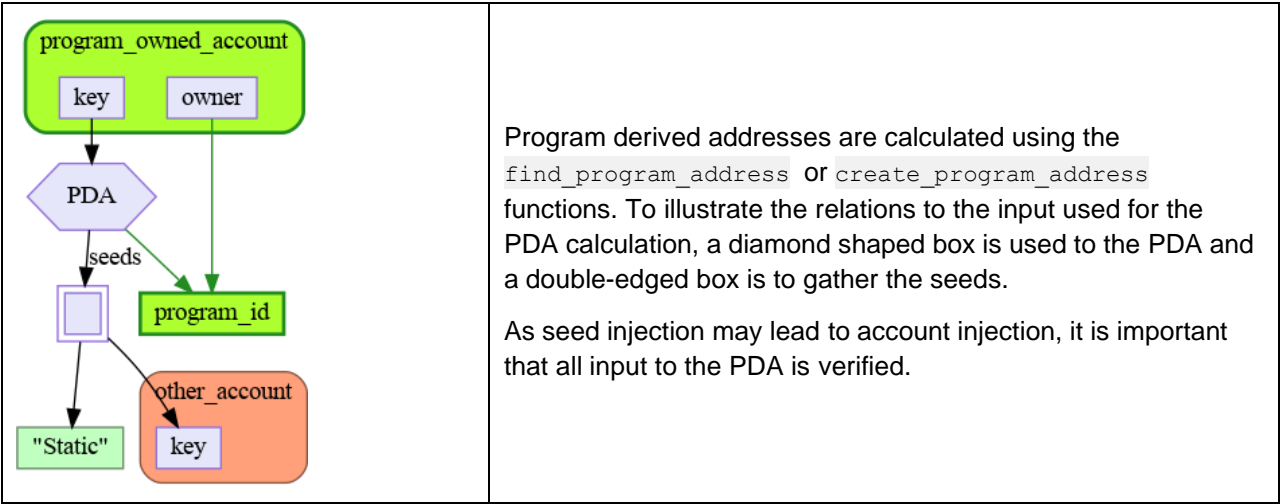
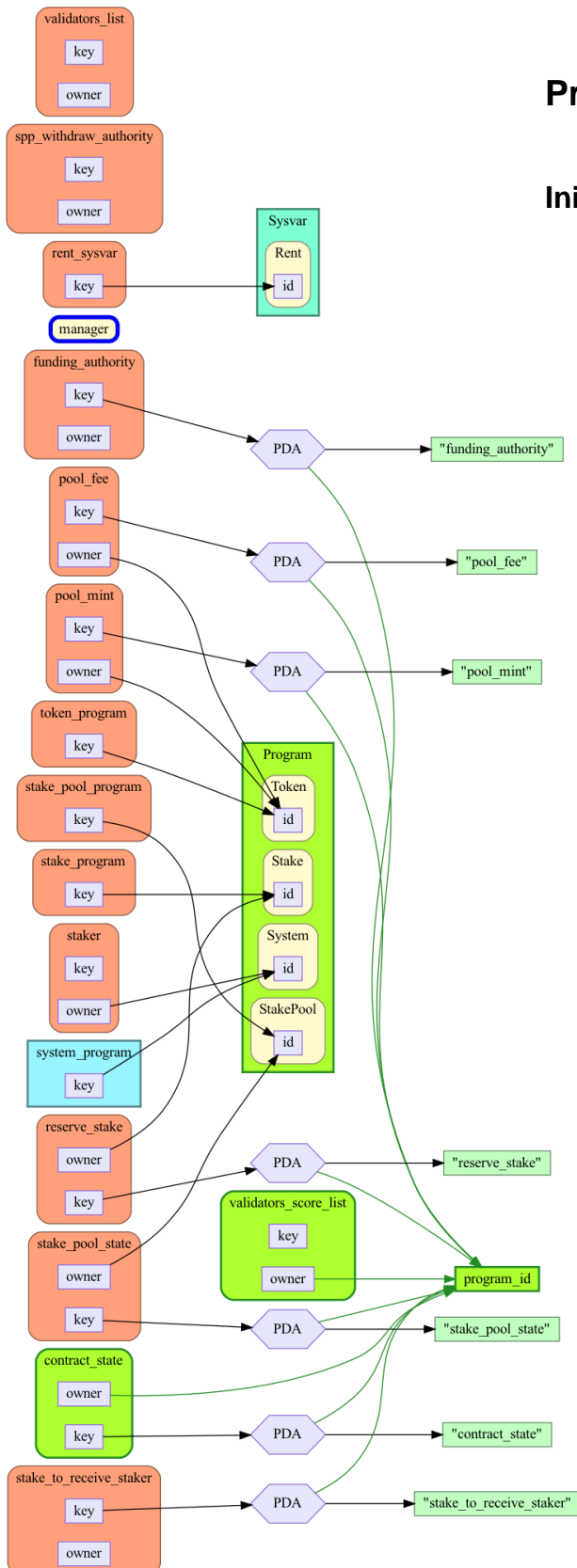


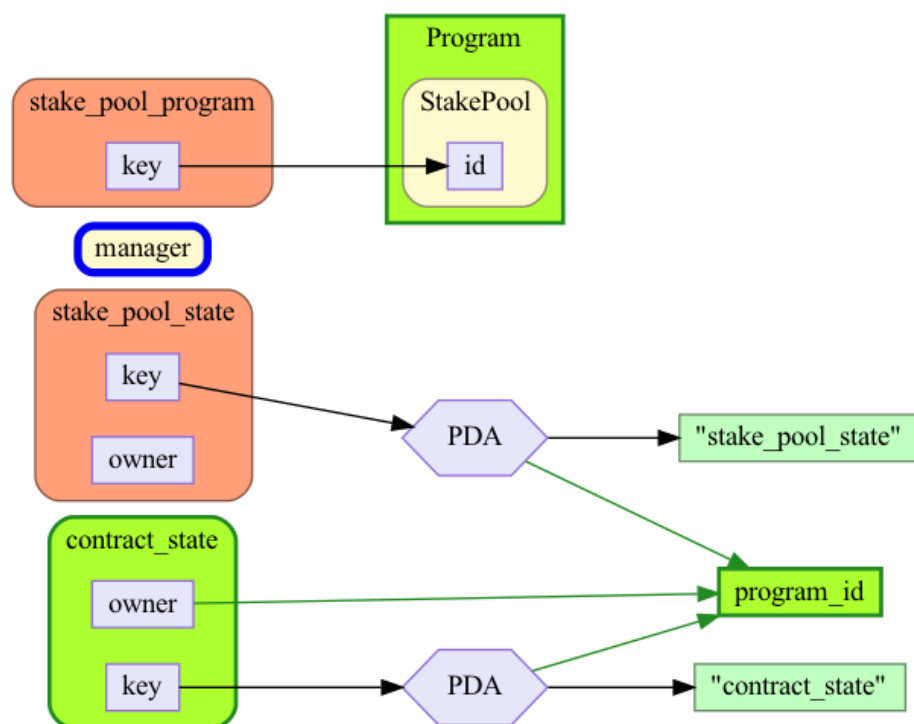
Table 3: Legend for relationship graphs

Privilege: Manager

Initialize



Update fee

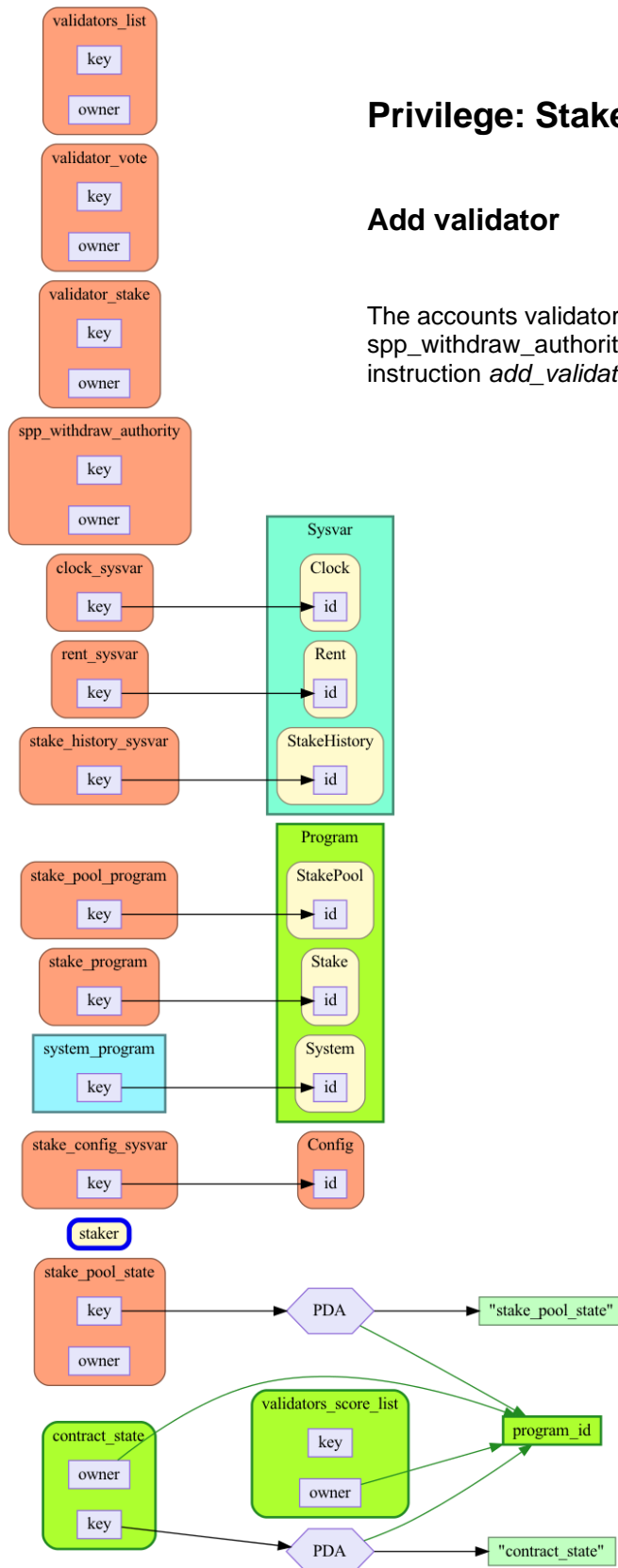


The owner of the stake_pool_state is not checked. It is, however, a PDA based on a constant seed, thus the check is not required.

Privilege: Staker

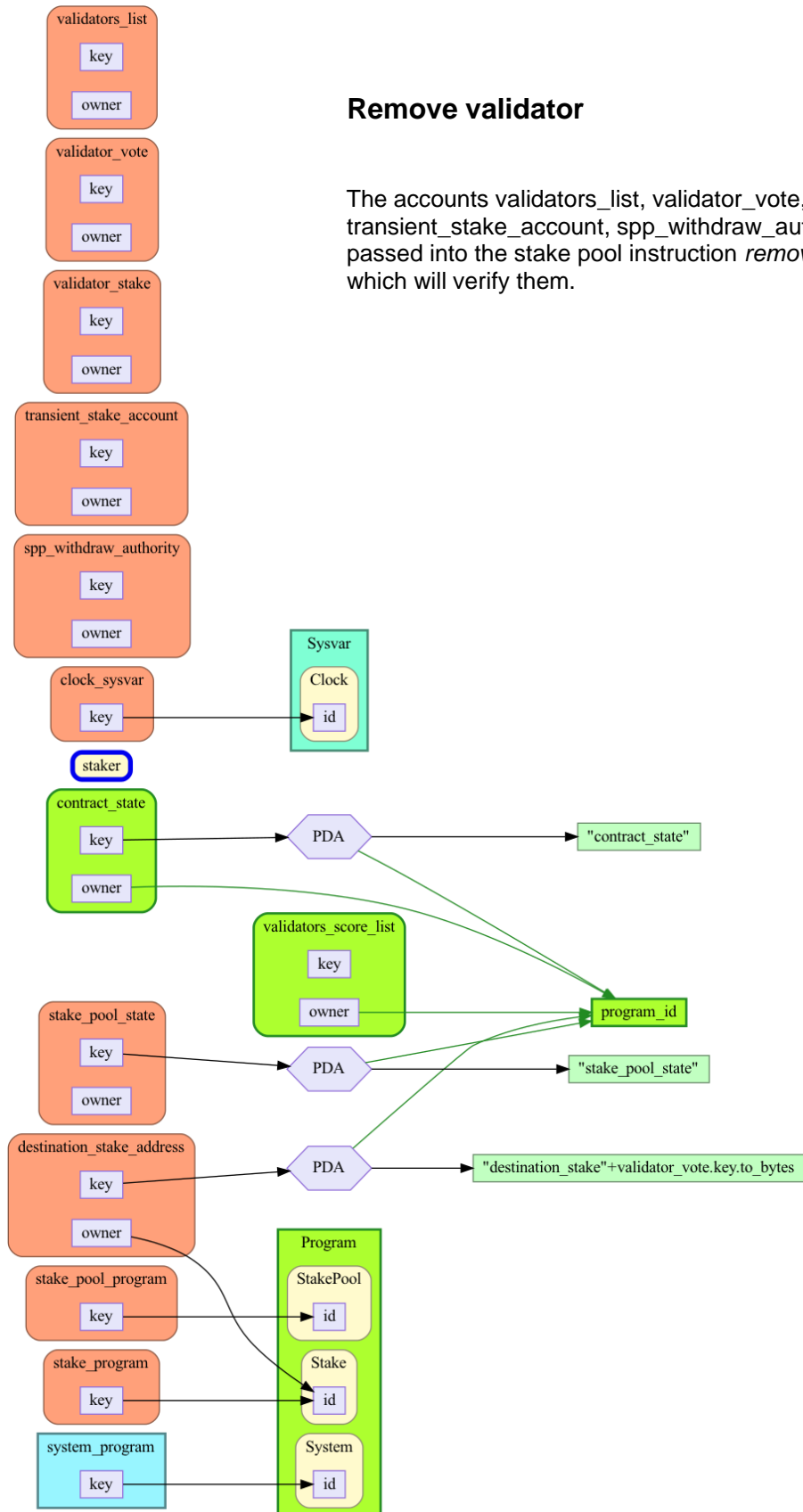
Add validator

The accounts `validators_list`, `validator_vote`, `validator_stake`, `spp_withdraw_authority` are unchecked, but they are passed into the stake pool instruction `add_validator_to_pool_with_vote` which will verify them.



Remove validator

The accounts `validators_list`, `validator_vote`, `validator_stake`, `transient_stake_account`, `spp_withdraw_authority` are unchecked, but they are passed into the stake pool instruction `remove_validator_from_pool_with_vote` which will verify them.

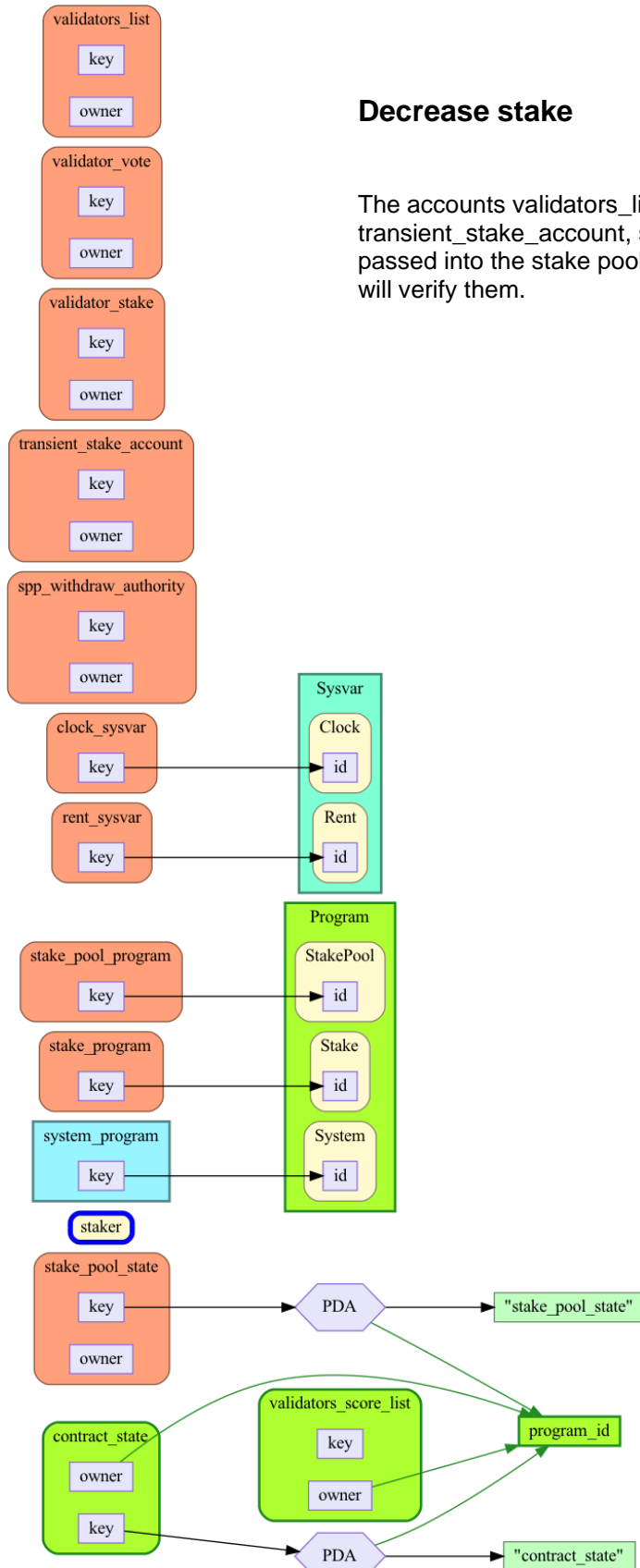




The accounts `validators_list`, `validator_vote`, `transient_stake_account`, `spp_withdraw_authority` and `stake_config_sysvar` are unchecked, but they are passed into the stake pool instruction `increase_validator_stake_with_vote` which will verify them.

Decrease stake

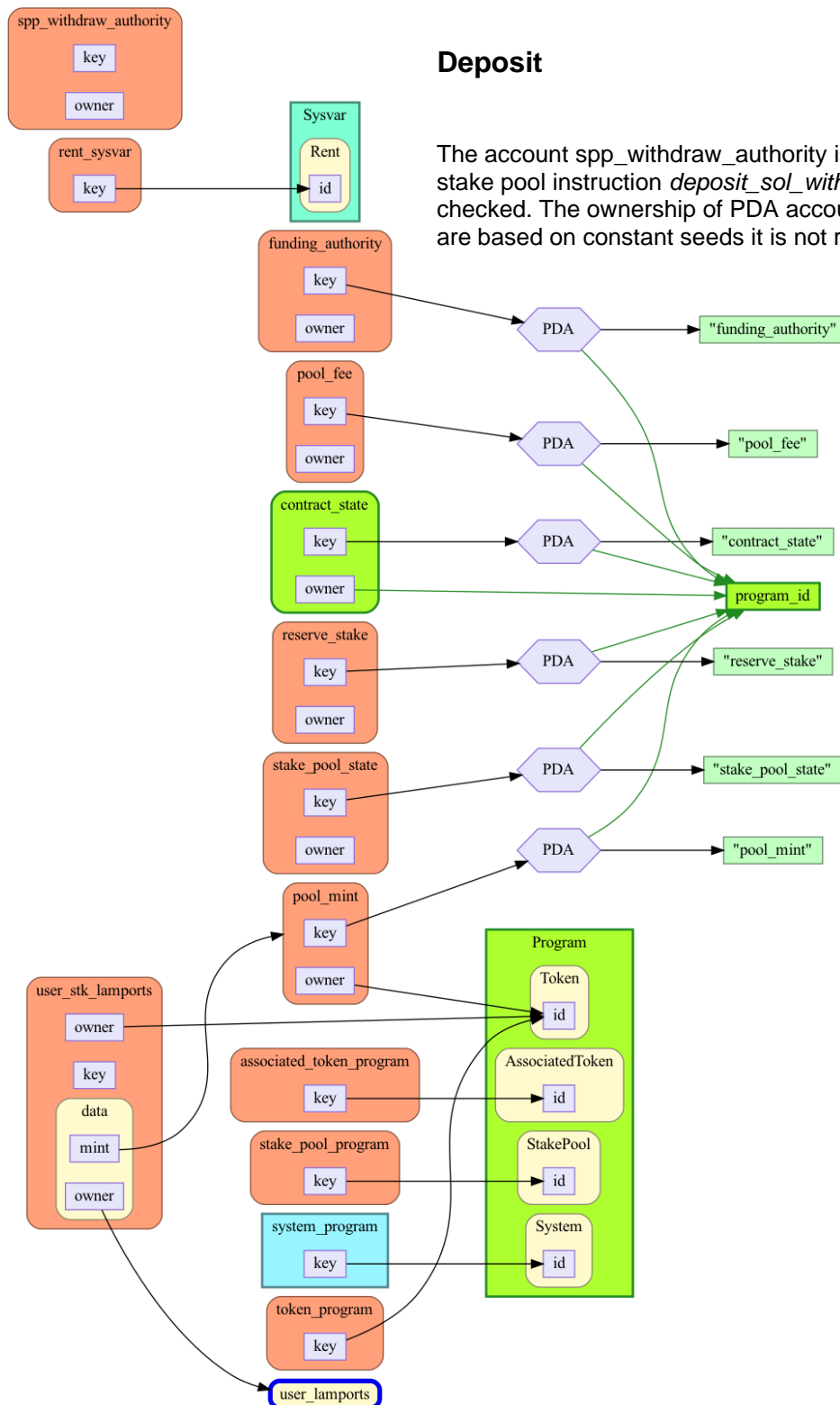
The accounts `validators_list`, `validator_vote`, `validator_stake`, `transient_stake_account`, `spp_withdraw_authority` are unchecked, but they are passed into the stake pool instruction `decrease_validator_stake_with_vote` which will verify them.



Privilege: User

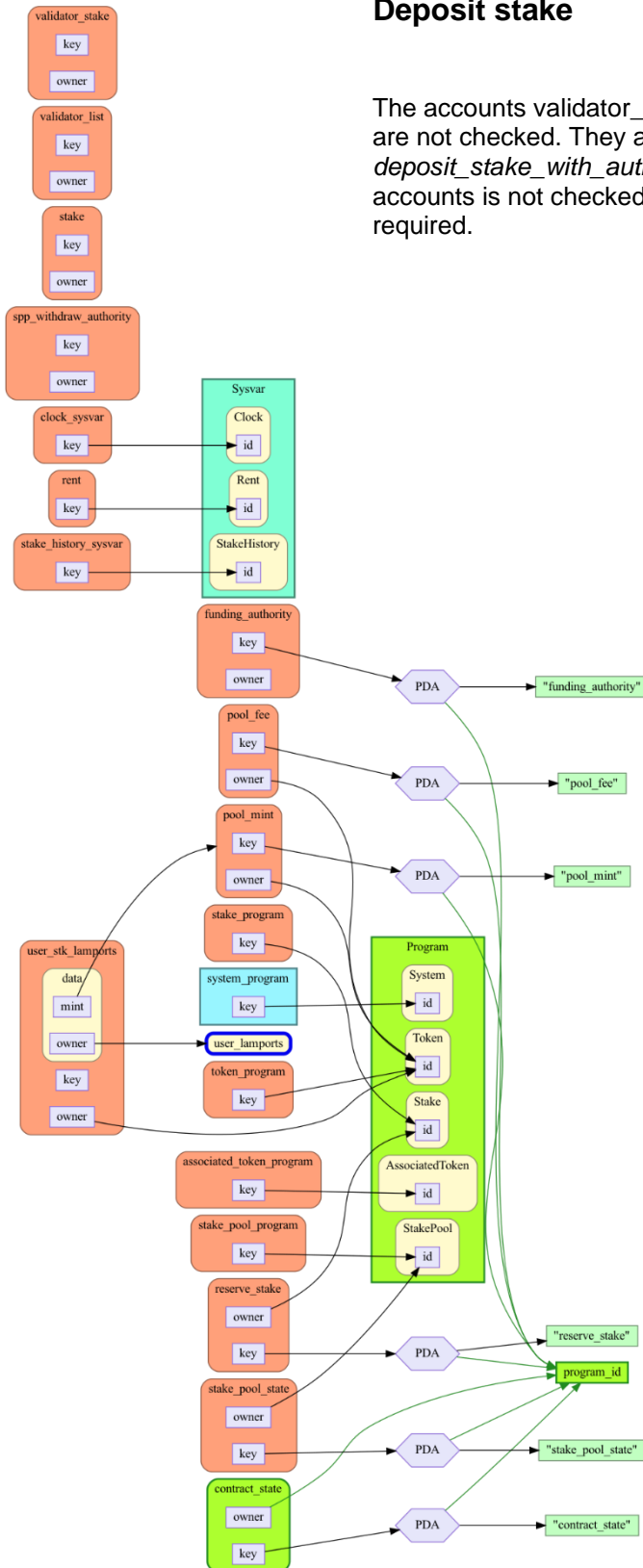
Deposit

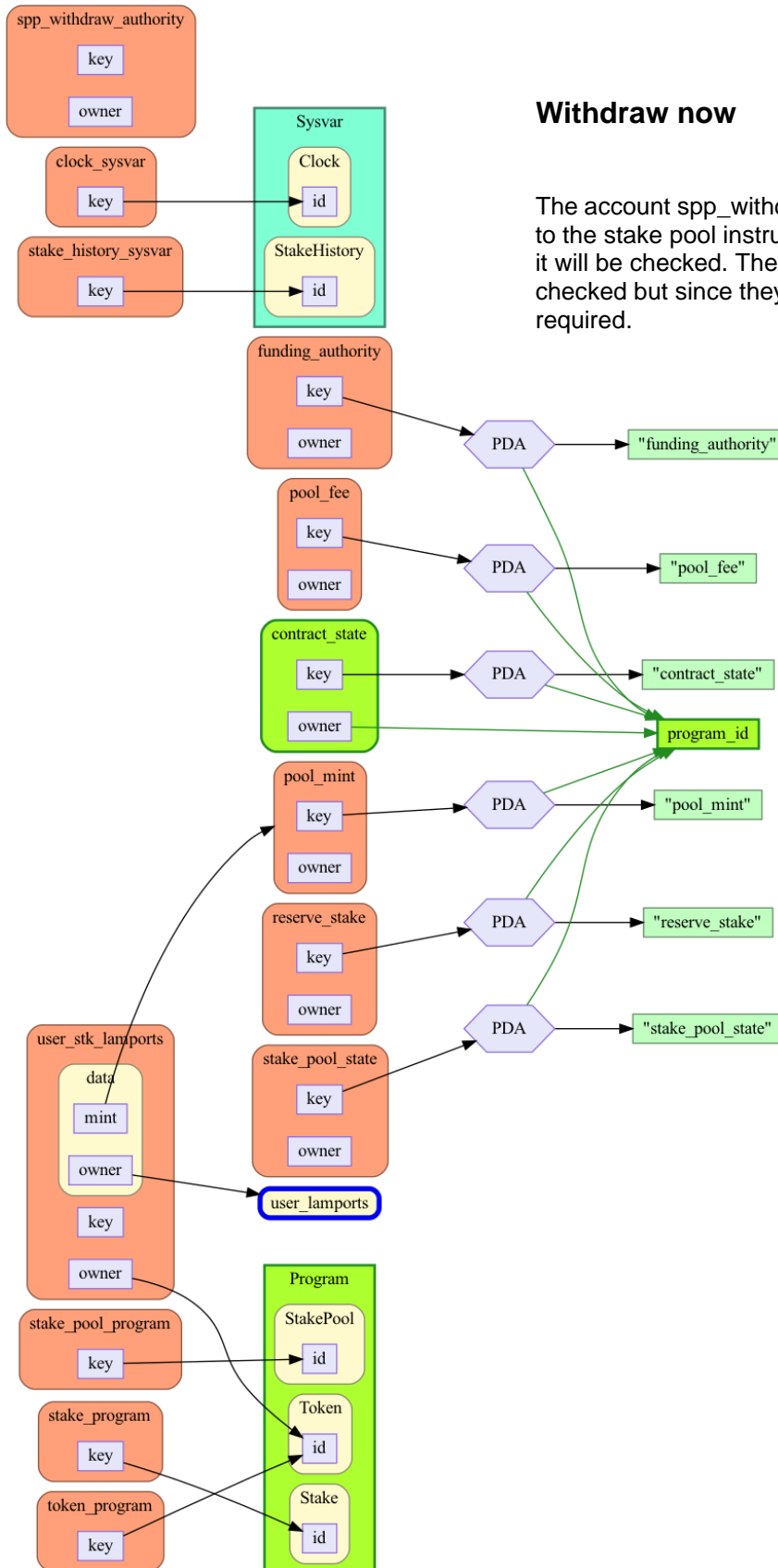
The account `spp_withdraw_authority` is not checked. It is passed to the stake pool instruction `deposit_sol_with_authority`, where it will be checked. The ownership of PDA accounts is not checked, but since they are based on constant seeds it is not required.



Deposit stake

The accounts `validator_stake`, `validator_list`, `stake` and `spp_withdraw_authority` are not checked. They are passed to the stake pool instruction `deposit_stake_with_authority`, where they will be checked. The ownership of PDA accounts is not checked, but since they are based on constant seeds it is not required.

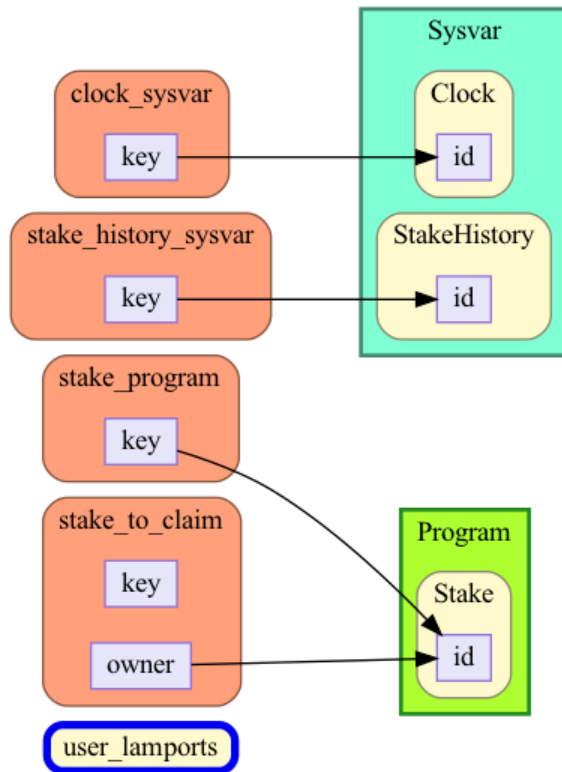




The accounts validators_list, stake_to_split and spp_withdraw_authority are not checked. They are passed to the stake pool instruction *withdraw_stake*, where they will be checked. The ownership of PDA accounts is not checked but since they are based on constant seeds it is not required.



Claim



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 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-reviewed 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
Ryan Spanier	VP, Global Innovation	Ryan.Spanier@kudelskisecurity.com
Shannon Garcia	Director Application Security	Shanon.Garcia@kudelskisecurity.com