Zellic

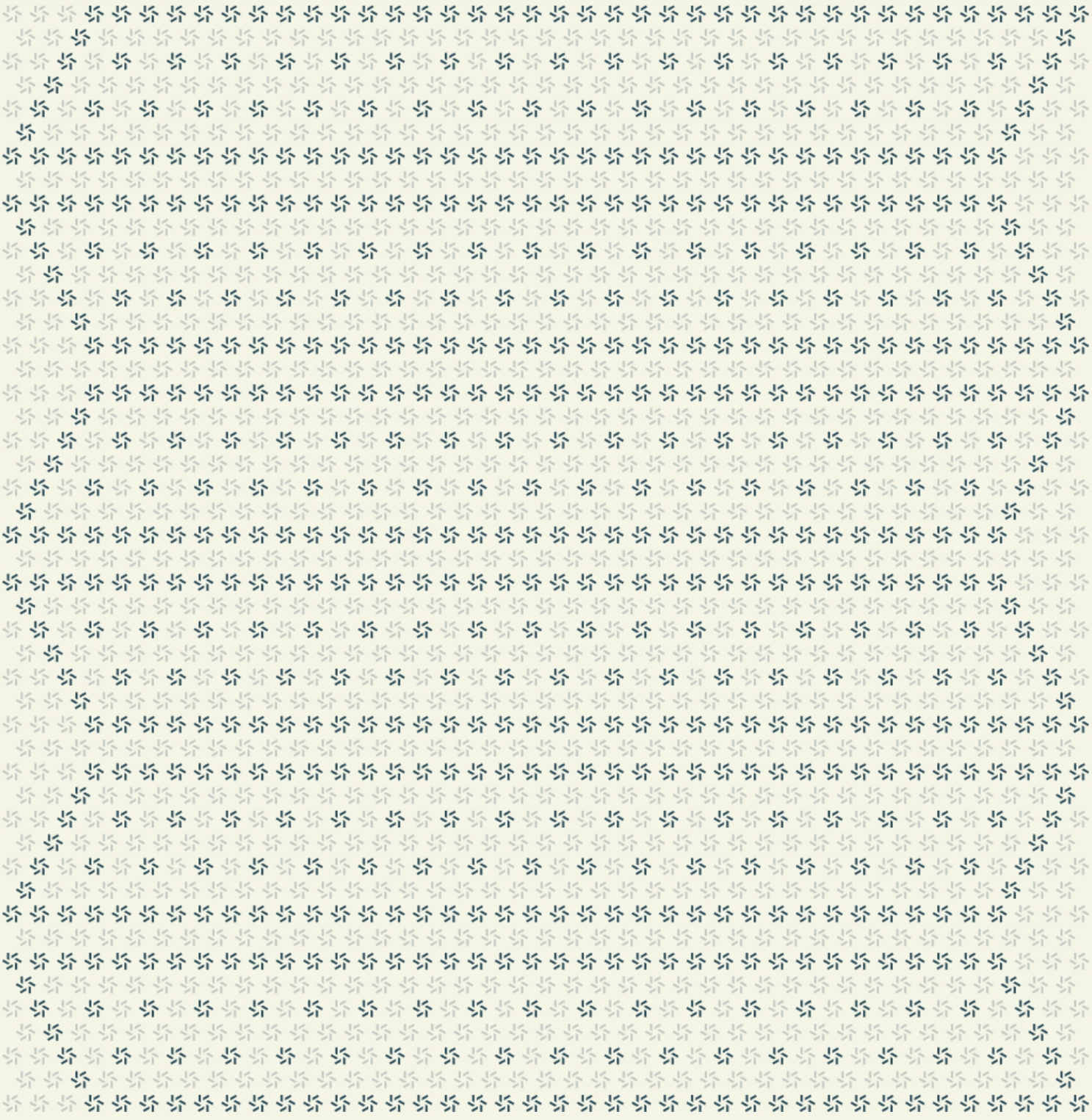
Prepared for
Hanako Mumei
Anza

Prepared by
Aaron Esau
Ethan Lee
Zellic

March 12, 2025

BPF Stake Program

Solana Application Security Assessment



Contents

About Zellic **3**

1. Overview **3**

1.1. Executive Summary 4

1.2. Goals of the Assessment 4

1.3. Non-goals and Limitations 4

1.4. Results 4

2. Introduction **5**

2.1. About BPF Stake Program 6

2.2. Methodology 6

2.3. Scope 8

2.4. Project Overview 8

2.5. Project Timeline 9

3. Discussion **9**

3.1. Test suite 10

4. Threat Model **11**

4.1. solana-stake program 13

5. Assessment Results **41**

5.1. Disclaimer 42

DRAFT

About Zellic

Zellic is a vulnerability research firm with deep expertise in blockchain security. We specialize in EVM, Move (Aptos and Sui), and Solana as well as Cairo, NEAR, and Cosmos. We review L1s and L2s, cross-chain protocols, wallets and applied cryptography, zero-knowledge circuits, web applications, and more.

Prior to Zellic, we founded the #1 CTF (competitive hacking) team ↗ worldwide in 2020, 2021, and 2023. Our engineers bring a rich set of skills and backgrounds, including cryptography, web security, mobile security, low-level exploitation, and finance. Our background in traditional information security and competitive hacking has enabled us to consistently discover hidden vulnerabilities and develop novel security research, earning us the reputation as the go-to security firm for teams whose rate of innovation outpaces the existing security landscape.

For more on Zellic's ongoing security research initiatives, check out our website zellic.io ↗ and follow @zellic_io ↗ on Twitter. If you are interested in partnering with Zellic, contact us at hello@zellic.io ↗.

DRAFT

1. Overview

1.1. Executive Summary

Zellic conducted a security assessment for Anza from February 21th to March 7th, 2025. During this engagement, Zellic reviewed BPF Stake Program's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.2. Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- Are all state changes properly done?
- Are signer checks in place where they belong?
- Are ownership checks in place where they belong?
- What edge-case differences exist between this implementation and the previous implementation?

1.3. Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- Front-end components
- Infrastructure relating to the project
- Key custody

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

1.4. Results

During our assessment on the scoped BPF Stake Program programs, there were no security vulnerabilities discovered.

Zellic recorded its notes and observations from the assessment for the benefit of Anza in the Discussion section (3. ↗).

DRAFT

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	0
Medium	0
Low	0
Informational	0

DRAFT

2. Introduction

2.1. About BPF Stake Program

Anza contributed the following description of BPF Stake Program:

> The BPF Stake Program is a port of the existing Native Stake Program to run as a true onchain program instead of a component of the validator client.

2.2. Methodology

During a security assessment, Zellic works through standard phases of security auditing, including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the programs.

Business logic errors. Business logic is the heart of any smart contract application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.

Integration risks. Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. Time permitting, we review external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.

Code maturity. We look for potential improvements in the codebase in general. We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradability weaknesses, centralization risks, and so on.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood.

DRAFT

We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

Finally, Zellic provides a list of miscellaneous observations that do not have security impact or are not directly related to the scoped programs itself. These observations — found in the Discussion (3. ↗) section of the document — may include suggestions for improving the codebase, or general recommendations, but do not necessarily convey that we suggest a code change.

DRAFT

2.3. Scope

The engagement involved a review of the following targets:

BPF Stake Program Programs

Type	Rust
Platform	Solana
Target	stake
Repository	https://github.com/solana-program/stake ↗
Version	`5ec49ccb08c3e588940a2038c99efc7acf563b4a`
Programs	`entrypoint.rs` `processor.rs`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of 2.2 person-weeks. The assessment was conducted by two consultants over the course of 11 calendar days.

DRAFT

Contact Information

The following project managers were associated with the engagement:

Jacob Goreski
Engagement Manager
jacob@zellic.io

Chad McDonald
Engagement Manager
chad@zellic.io

The following consultants were engaged to conduct the assessment:

Aaron Esau
Engineer
aaron@zellic.io

Ethan Lee
Engineer
ethl@zellic.io

2.5. Project Timeline

The key dates of the engagement are detailed below.

February 20, 2025	Kick-off call
February 21, 2025	Start of primary review period
March 7, 2025	End of primary review period

DRAFT

3. Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment. These discussion notes are not necessarily security related and do not convey that we are suggesting a code change.

3.1. Test suite

When building a complex contract ecosystem with multiple moving parts and dependencies, comprehensive testing is essential. This includes testing for both positive and negative scenarios. Positive tests should verify that each function's side effect is as expected, while negative tests should cover every revert, preferably in every logical branch.

Good test coverage has multiple effects.

- It finds bugs and design flaws early (preaudit or prerelease).
- It gives insight into areas for optimization (e.g., gas cost).
- It displays code maturity.
- It improves understanding of how the code functions, integrates, and operates — for developers and auditors alike.

We recommend creating unit tests for the following instructions:

- `StakeInstruction::SetLockup`
- `StakeInstruction::AuthorizeWithSeed`
- `StakeInstruction::DeactivateDelinquent`

Additionally, we believe the following tests should be more comprehensive; that is, the following tests do not cover all positive or negative behaviors:

- `StakeInstruction::SetLockupChecked`
  - ☑ Positive test — Expected behavior when the stake account's state is `StakeStateV2::Initialized`. Note that the current test does not assert that the proper state changes have been made to the account lockup.
  - ☐ Positive test — Expected behavior when the stake account's state is `StakeStateV2::Stake`.
  - ☑ Negative test — Missing required signatures.
  - ☐ Negative test — The stake account's state is not `StakeStateV2::Initialized` or `StakeStateV2::Stake`.
  - ☐ Negative test — The lockup is in force.
- `StakeInstruction::AuthorizeCheckedWithSeed`
  - ☑ Positive test — Expected behavior when the stake account's state is `StakeStateV2::Initialized`. Note that the current test does not assert that the proper state changes have been made to the authorized staker or withdrawer.

DRAFT

- ☐ Positive test — Expected behavior when the stake account's state is `StakeStateV2::Stake`.
- ☑ Negative test — Missing required signatures.
- ☐ Negative test — The stake account's state is not `StakeStateV2::Initialized` or `StakeStateV2::Stake`.
- ☐ Negative test — The lockup is in force.

Practically speaking, most of the code in the underlying state.rs module (used by the `StakeInstruction::SetLockupChecked` and `StakeInstruction::AuthorizeCheckedWithSeed` functions in particular) has been tested in the staking program tests. However, we recommend testing through the new BPF interface too for the aforementioned reasons.

DRAFT

4. Threat Model

As time permitted, we analyzed each instruction in the program and created a written threat model for the most critical instructions. A threat model documents the high-level functionality of a given instruction, the inputs it receives, and the accounts it operates on as well as the main checks performed on them; it gives an overview of the attack surface of the programs and of the level of control an attacker has over the inputs of critical instructions.

For brevity, system accounts and well-known program accounts have not been included in the list of accounts received by an instruction; the instructions that receive these accounts make use of Anchor types, which automatically ensure that the public key of the account is correct.

Not all instructions in the audit scope may have been modeled. The absence of a threat model in this section does not necessarily suggest that an instruction is safe.

DRAFT

4.1. solana-stake program

Instruction: `Authorize`

This instruction updates the stake or withdraw authority of a stake account who manages stake or withdrawal.

Input structure

```
pub enum StakeAuthorize {
    Staker,
    Withdrawer,
}

pub enum StakeInstruction {
    /// # Account references
    ///   0. `[WRITE]` Stake account to be updated
    ///   1. `[]` Clock sysvar
    ///   2. `[SIGNER]` The stake or withdraw authority
    ///   3. Optional: `[SIGNER]` Lockup authority, if updating
    StakeAuthorize::Withdrawer before
    ///      lockup expiration
    Authorize(Pubkey, StakeAuthorize),
}

fn process_authorize(
    accounts: &[AccountInfo],
    new_authority: Pubkey,
    authority_type: StakeAuthorize,
) -> ProgramResult {
```

Parameters

- `new_authority`: The new authority to update.
- `authority_type`: The type of authority to update.

Accounts

- **`stake_account`**: The stake account to be updated.
  - Signer: No.
  - Init: No.

    - PDA: No.
    - Writable: Yes.
    - Rent checks: None.
    - Ownership checks: The account must be owned by the program.
    - Address checks: None.
- **`clock`**: The account must be a clock sysvar.
- **`stake_or_withdraw_authority`**: Unused.
- **`option_lockup_authority`** (optional): The lockup authority account, if updating `StakeAuthorize::Withdrawer` before lockup expiration.
    - Signer: Yes.
    - Init: No.
    - PDA: No.
    - Writable: No.
    - Rent checks: None.
    - Ownership checks: None.
    - Address checks: None.

Additional checks and behavior

- `lockup_authority` must be a signer if provided.
- When updating the staker, either the current staker or the withdrawer of the stake account must be a signer.
- When updating the withdrawer, the current withdrawer of the stake account must be a signer.
- When updating the withdrawer, the lockup must not be in force or the custodian must be a signer.
- The stake account authority is updated with the provided `new_authorized` key.

Instruction: `AuthorizeChecked`

This instruction updates the stake or withdraw authority of a stake account who manages stake or withdrawal. This instruction behaves like `Authorize` with the additional requirement that the new authority must be a signer.

Input structure

```
pub enum StakeAuthorize {
    Staker,
    Withdrawer,
```

```
}

pub enum StakeInstruction {
    /// # Account references
    ///   0. `[WRITE]` Stake account to be updated
    ///   1. `[]` Clock sysvar
    ///   2. `[SIGNER]` The stake or withdraw authority
    ///   3. `[SIGNER]` The new stake or withdraw authority
    ///   4. Optional: `[SIGNER]` Lockup authority, if updating
    StakeAuthorize::Withdrawer before
    ///      lockup expiration
    AuthorizeChecked(StakeAuthorize),
}

fn process_authorize_checked(
    accounts: &[AccountInfo],
    authority_type: StakeAuthorize,
) -> ProgramResult {
```

Parameters

- `authority_type`: The type of authority to update.

Accounts

- **`stake_account`**: The stake account to be updated.
  - Signer: No.
  - Init: No.
  - PDA: No.
  - Writable: Yes.
  - Rent checks: None.
  - Ownership checks: The account must be owned by the program.
  - Address checks: None.
- **`clock`**: The account must be a clock sysvar.
- **`old_stake_or_withdraw_authority`**: Unused.
- **`new_stake_or_withdraw_authority`**: The new stake or withdraw authority account.
  - Signer: Yes.
  - Init: No.
  - PDA: No.
  - Writable: No.

- Rent checks: None.
- Ownership checks: None.
- Address checks: None.

- **`option_lockup_authority`** (optional): The lockup authority account, if updating `StakeAuthorize::Withdrawer` before lockup expiration.
  - Signer: Yes.
  - Init: No.
  - PDA: No.
  - Writable: No.
  - Rent checks: None.
  - Ownership checks: None.
  - Address checks: None.

Additional checks and behavior

- `lockup_authority` must be a signer if provided.
- `new_stake_or_withdraw_authority` must be a signer.
- When updating the staker, either the current staker or the withdrawer of the stake account must be a signer.
- When updating the withdrawer, the current withdrawer of the stake account must be a signer.
- When updating the withdrawer, the lockup must not be in force or the custodian must be a signer.
- The stake account authority is updated with the provided `new_authorized` key.

Instruction: `AuthorizeCheckedWithSeed`

This instruction updates the stake or withdraw authority of a stake account who manages stake or withdrawal. The stake or withdraw authority must be a signer to use an account derived from the seed as a signer. This instruction behaves like `AuthorizeWithSeed` with the additional requirement that the new authority must be a signer.

Input structure

```
pub enum StakeAuthorize {
    Staker,
    Withdrawer,
}

pub struct AuthorizeCheckedWithSeedArgs {
```

```
    pub stake_authorize: StakeAuthorize,
    pub authority_seed: String,
    pub authority_owner: Pubkey,
}

pub enum StakeInstruction {
    /// # Account references
    ///   0. `[WRITE]` Stake account to be updated
    ///   1. `[SIGNER]` Base key of stake or withdraw authority
    ///   2. `[]` Clock sysvar
    ///   3. `[SIGNER]` The new stake or withdraw authority
    ///   4. Optional: `[SIGNER]` Lockup authority, if updating
    StakeAuthorize::Withdrawer before
    ///      lockup expiration
    AuthorizeCheckedWithSeed(AuthorizeCheckedWithSeedArgs),
}
```

Parameters

- `stake_authorize`: The type of authority to update.
- `authority_seed`: The seed for the derived address.
- `authority_owner`: The owner of the derived address.

Accounts

- **`stake_account`**: The stake account to be updated.
  - Signer: No.
  - Init: No.
  - PDA: No.
  - Writable: Yes.
  - Rent checks: None.
  - Ownership checks: The account must be owned by the program.
  - Address checks: None.
- **`old_stake_or_withdraw_authority_base`**: The base key of the stake or withdraw authority.
  - Signer: Yes.
  - Init: No.
  - PDA: No.
  - Writable: No.
  - Rent checks: None.
  - Ownership checks: None.

DRAFT

    - Address checks: None.
- **`clock`**: The account must be a clock sysvar.
- **`new_stake_or_withdraw_authority`**: The new stake or withdraw authority account.
    - Signer: Yes.
    - Init: No.
    - PDA: No.
    - Writable: No.
    - Rent checks: None.
    - Ownership checks: None.
    - Address checks: None.
- **`option_lockup_authority`** (optional): The lockup authority account, if updating `StakeAuthorize::Withdrawer` before lockup expiration.
    - Signer: Yes.
    - Init: No.
    - PDA: No.
    - Writable: No.
    - Rent checks: None.
    - Ownership checks: None.
    - Address checks: None.

Additional checks and behavior

- `lockup_authority` must be a signer if provided.
- `new_stake_or_withdraw_authority` must be a signer.
- `stake_or_withdraw_authority_base` must be a signer so that the account derived from the seed becomes a signer.
- When updating the staker, either the current staker or the withdrawer of the stake account must be a signer.
- When updating the withdrawer, the current withdrawer of the stake account must be a signer.
- When updating the withdrawer, the lockup must not be in force or the custodian must be a signer.
- The stake account authority is updated with the provided `new_authorized` key.

Instruction: `AuthorizeWithSeed`

This instruction updates the stake or withdraw authority of a stake account who manages stake or withdrawal. The stake or withdraw authority must be a signer to use an account derived from the seed as a signer.

DRAFT

Input structure

```
pub struct AuthorizeWithSeedArgs {
    pub new_authorized_pubkey: Pubkey,
    pub stake_authorize: StakeAuthorize,
    pub authority_seed: String,
    pub authority_owner: Pubkey,
}

pub enum StakeInstruction {
    /// # Account references
    ///   0. `[WRITE]` Stake account to be updated
    ///   1. `[SIGNER]` Base key of stake or withdraw authority
    ///   2. `[]` Clock sysvar
    ///   3. Optional: `[SIGNER]` Lockup authority, if updating
    StakeAuthorize::Withdrawer before
    ///      lockup expiration
    AuthorizeWithSeed(AuthorizeWithSeedArgs),
}

fn process_authorize_with_seed(
    accounts: &[AccountInfo],
    authorize_args: AuthorizeWithSeedArgs,
) -> ProgramResult {
```

Parameters

- `new_authorized_pubkey`: The new authority to update.
- `stake_authorize`: The type of authority to update.
- `authority_seed`: The seed for the derived address.
- `authority_owner`: The owner of the derived address.

Accounts

- **`stake_account`**: The stake account to be updated.
  - Signer: No.
  - Init: No.
  - PDA: No.
  - Writable: Yes.
  - Rent checks: None.
  - Ownership checks: The account must be owned by the program.

    - Address checks: None.
- **`clock`**: The account must be a clock sysvar.
- **`stake_or_withdraw_authority_base`**: The base key of the stake or withdraw authority.
    - Signer: Yes.
    - Init: No.
    - PDA: No.
    - Writable: No.
    - Rent checks: None.
    - Ownership checks: None.
    - Address checks: None.
- **`option_lockup_authority`** (optional): The lockup authority account, if updating `StakeAuthorize::Withdrawer` before lockup expiration.
    - Signer: Yes.
    - Init: No.
    - PDA: No.
    - Writable: No.
    - Rent checks: None.
    - Ownership checks: None.
    - Address checks: None.

Additional checks and behavior

- `lockup_authority` must be a signer if provided.
- `stake_or_withdraw_authority_base` must be a signer so that the account derived from the seed becomes a signer.
- When updating the staker, either the current staker or the withdrawer of the stake account must be a signer.
- When updating the withdrawer, the current withdrawer of the stake account must be a signer.
- When updating the withdrawer, the lockup must not be in force or the custodian must be a signer.
- The stake account authority is updated with the provided `new_authorized` key.

Instruction: `Deactivate`

This instruction deactivates delegation of a stake account by setting the deactivation epoch.

DRAFT

Input structure

```
pub enum StakeInstruction {
    /// # Account references
    ///   0. `[WRITE]` Delegated stake account
    ///   1. `[]` Clock sysvar
    ///   2. `[SIGNER]` Stake authority
    Deactivate,
}
```

Accounts

- **`stake_account`**: The delegated stake account.
    - Signer: No.
    - Init: No.
    - PDA: No.
    - Writable: Yes.
    - Rent checks: None.
    - Ownership checks: The account must be owned by the program.
    - Address checks: None.
- **`clock`**: The account must be a clock sysvar.
- **`stake_authority`**: The stake authority account who is authorized to manage the stake account.
    - Signer: Yes.
    - Init: No.
    - PDA: No.
    - Writable: No.
    - Rent checks: None.
    - Ownership checks: None.
    - Address checks: None.

Additional checks and behavior

- The staker of the stake account must be a signer.
- The stake account must be in the `Stake` state.
- The stake account must not be deactivated.
- The deactivation epoch of the stake account is set to the current epoch.

DRAFT

Instruction: `DeactivateDelinquent`

This instruction deactivates delegation of a stake account that has been delegated to a vote account that has been delinquent for at least `MINIMUM_DELINQUENT_EPOCHS_FOR_DEACTIVATION` epochs.

Input structure

```
pub enum StakeInstruction {
    /// # Account references
    ///   0. `[WRITE]` Delegated stake account
    ///   1. `[]` Delinquent vote account for the delegated stake account
    ///   2. `[]` Reference vote account that has voted at least once in the
    last
    ///      `MINIMUM_DELINQUENT_EPOCHS_FOR_DEACTIVATION` epochs
    DeactivateDelinquent,
}
```

Accounts

- **`stake_account`**: The delegated stake account to be deactivated.
  - Signer: No.
  - Init: No.
  - PDA: No.
  - Writable: Yes.
  - Rent checks: None.
  - Ownership checks: The account must be owned by the program.
  - Address checks: None.
- **`delinquent_vote_account`**: The delinquent vote account for the delegated stake account.
  - Signer: No.
  - Init: No.
  - PDA: No.
  - Writable: No.
  - Rent checks: None.
  - Ownership checks: The account must be owned by the Solana vote program.
  - Address checks: None.
- **`reference_vote_account`**: The reference vote account that is active over previous `MINIMUM_DELINQUENT_EPOCHS_FOR_DEACTIVATION` epochs.

DRAFT

- Signer: No.
- Init: No.
- PDA: No.
- Writable: No.
- Rent checks: None.
- Ownership checks: The account must be owned by the Solana vote program.
- Address checks: None.

Additional checks and behavior

- The `reference_vote_account` must have been activated over the previous `MINIMUM_DELINQUENT_EPOCHS_FOR_DEACTIVATION` epochs.
- The stake account must be in the `Stake` state.
- The stake account must be delegated to the `delinquent_vote_account`.
- The `delinquent_vote_account` must have been delinquent over the previous `MINIMUM_DELINQUENT_EPOCHS_FOR_DEACTIVATION` epochs.
- Delegation of the stake account is deactivated if the above conditions are met.

Instruction: `DelegateStake`

This instruction delegates a stake account to a vote account. If the stake account is already delegated, the stake account is redelegated to the provided vote account.

Input structure

```
pub enum StakeInstruction {
    /// # Account references
    ///   0. `[WRITE]` Initialized stake account to be delegated
    ///   1. `[]` Vote account to which this stake will be delegated
    ///   2. `[]` Clock sysvar
    ///   3. `[]` Stake history sysvar that carries stake warmup/cooldown
    history
    ///   4. `[]` Unused account, formerly the stake config
    ///   5. `[SIGNER]` Stake authority
    DelegateStake,
}
```

Accounts

- **`stake_account`**: The initialized stake account to be delegated.

    - Signer: No.
    - Init: No.
    - PDA: No.
    - Writable: Yes.
    - Rent checks: The account must be rent-exempt.
    - Ownership checks: The account must be owned by the program.
    - Address checks: None.
- **`vote_account`**: The vote account to which this stake will be delegated.
    - Signer: No.
    - Init: No.
    - PDA: No.
    - Writable: Yes.
    - Rent checks: None.
    - Ownership checks: The account must be owned by the Solana vote program.
    - Address checks: None.
- **`clock`**: The account must be a clock sysvar.
- **`stake_history_info`**: Unused.
- **`stake_config`**: Unused.
- **`stake_authority`**: The stake authority account who is authorized to manage the stake account.
    - Signer: Yes.
    - Init: No.
    - PDA: No.
    - Writable: No.
    - Rent checks: None.
    - Ownership checks: None.
    - Address checks: None.

Additional checks and behavior

- The stake account must be initialized or delegated.
- The current staker or the withdrawer of the stake account must be a signer.
- The stake account must have enough lamports to be rent-exempt.
- The stake amount must be greater than the minimum delegation amount.
- The stake account state is set to `Stake` with the provided stake amount and vote account if the stake account is initialized.
- If the stake account is already delegated, the stake account is redelegated to the provided vote account.
    - If the stake is active, deactivation is rescinded if the new voter pubkey is the

DRAFT

same as the current voter pubkey and the stake is scheduled to start deactivating this epoch. Otherwise, an error is returned, indicating that it is too soon to redelegate.
- The activation epoch of the stake delegation is set to the current epoch.
- The deactivation epoch of the stake delegation is set to `u64::MAX` to indicate that the stake is not deactivating.
- The voter of the stake delegation is set to the provided vote account pubkey.
- The credits observed are set to the credits of the provided vote account.

Instruction: `GetMinimumDelegation`

This instruction returns the minimum stake amount that can be delegated, in lamports.

Input structure

```
pub enum StakeInstruction {
    /// # Account references
    ///   None
    GetMinimumDelegation
}
```

Accounts

- None.

Additional checks and behavior

- None.

Instruction: `Initialize`

This instruction initializes an uninitialized stake account with authorization and lockup information.

Input structure

```
pub struct Authorized {
    pub staker: Pubkey,
    pub withdrawer: Pubkey,
```

```
}

pub struct Lockup {
    /// UnixTimestamp at which this stake will allow withdrawal, unless the
    ///   transaction is signed by the custodian
    pub unix_timestamp: UnixTimestamp,
    /// epoch height at which this stake will allow withdrawal, unless the
    ///   transaction is signed by the custodian
    pub epoch: Epoch,
    /// custodian signature on a transaction exempts the operation from
    ///  lockup constraints
    pub custodian: Pubkey,
}

pub enum StakeInstruction {
    /// # Account references
    ///   0. `[WRITE]` Uninitialized stake account
    ///   1. `[]` Rent sysvar
    Initialize(Authorized, Lockup)
}

fn process_initialize(
    accounts: &[AccountInfo],
    authorized: Authorized,
    lockup: Lockup,
) -> ProgramResult {
```

Parameters

- `authorized`: The staker and withdrawer to be authorized.
- `lockup`: The lockup information.

Lockup

- `unix_timestamp`: Unix timestamp when the lockup expires.
- `epoch`: Epoch when the lockup expires.
- `custodian`: Custodian key on a transaction exempts the operation from lockup constraints.

- `authorized`: The staker and withdrawer to be authorized.
- `lockup`: The lockup information.

Accounts

- **`stake_account`**: The uninitialized stake account.

    - Signer: No.
    - Init: No.
    - PDA: No.
    - Writable: Yes.
    - Rent checks: The account must be rent-exempt.
    - Ownership checks: The account must be owned by the program.
    - Address checks: None.
- **`rent`**: The account must be a rent sysvar.

Additional checks and behavior

- The size of the stake account must be equal to the size of `StakeStateV2`.
- The stake account must be uninitialized.
- The stake account must have enough lamports to be rent-exempt.
- The stake account is initialized with the provided `authorized` and `lockup` information.

Instruction: `InitializeChecked`

This instruction initializes an uninitialized stake account with authorization and without lockup information. This instruction behaves like `Initialize` with the additional requirement that the withdraw authority must be a signer.

Input structure

```
pub enum StakeInstruction {
    /// # Account references
    ///   0. `[WRITE]` Uninitialized stake account
    ///   1. `[]` Rent sysvar
    ///   2. `[]` The stake authority
    ///   3. `[SIGNER]` The withdraw authority
    InitializeChecked,
}
```

Accounts

- **`stake_account`**: The uninitialized stake account.
    - Signer: No.
    - Init: No.
    - PDA: No.

    - Writable: Yes.
    - Rent checks: The account must be rent-exempt.
    - Ownership checks: The account must be owned by the program.
    - Address checks: None.
- **`rent`**: The account must be a rent sysvar.
- **`stake_authority`**: The stake authority account to be authorized to manage the stake account.
    - Signer: No.
    - Init: No.
    - PDA: No.
    - Writable: No.
    - Rent checks: None.
    - Ownership checks: None.
    - Address checks: None.
- **`withdraw_authority`**: The withdraw authority account to be authorized to withdraw from the stake account.
    - Signer: Yes.
    - Init: No.
    - PDA: No.
    - Writable: No.
    - Rent checks: None.
    - Ownership checks: None.
    - Address checks: None.

Additional checks and behavior

- The size of `stake_account` must be equal to the size of `StakeStateV2`.
- `withdraw_authority` must be a signer.
- The stake account must be uninitialized.
- The stake account must have enough lamports to be rent-exempt.
- The stake account is initialized with the provided `authorized` and `lockup` information.

Instruction: `Merge`

This instruction splits tokens and stake off a stake account into a new stake account.

DRAFT

Input structure

```
pub enum StakeInstruction {
    /// # Account references
    ///   0. `[WRITE]` Destination stake account for the merge
    ///   1. `[WRITE]` Source stake account for to merge.  This account will be
    drained
    ///   2. `[]` Clock sysvar
    ///   3. `[]` Stake history sysvar that carries stake warmup/cooldown
    history
    ///   4. `[SIGNER]` Stake authority
    Merge,
}
```

Accounts

- **`destination_stake_account`**: The destination stake account for the merge.
  - Signer: No.
  - Init: No.
  - PDA: No.
  - Writable: Yes.
  - Rent checks: None.
  - Ownership checks: The account must be owned by the program.
  - Address checks: The address must be different from the source stake account.
- **`source_stake_account`**: The source stake account to merge that will be drained.
  - Signer: No.
  - Init: No.
  - PDA: No.
  - Writable: Yes.
  - Rent checks: None.
  - Ownership checks: The account must be owned by the program.
  - Address checks: The address must be different from the destination stake account.
- **`clock`**: The account must be a clock sysvar.
- **`stake_history_info`**: Unused.
- **`stake_authority`**: The stake authority account who is authorized to manage the stake account.

DRAFT

- Signer: Yes.
- Init: No.
- PDA: No.
- Writable: No.
- Rent checks: None.
- Ownership checks: None.
- Address checks: None.

Additional checks and behavior

- The stake accounts must be in the `Stake` or `Initialized` state.
- The stake accounts must not be in a transient state, such as activating or deactivating with nonzero effective stake.
- The staker of the destination stake account must be the signer.
- The lockups of the source and destination stake accounts must match if they are in force.
- The authorities of the source and destination stake accounts must match.
- The stake accounts must have the same voter pubkey.
- Both stake accounts must not be scheduled for deactivation.
- If either merge kind of stake accounts is inactive, the destination stake account must be activating.
- If the destination stake account is activating and the source stake account is inactive, merge the source stake account into the destination stake account.
- If both stake accounts are activating, merge the source stake account into the destination stake account with rent-exempt reserve and update the credits observed.
- If both stake accounts are active, merge the source stake account into the destination stake account and update the credits observed.
- The source stake account state is deinitialized.
- All lamports are moved from the source stake account to the destination stake account.

Instruction: `MoveLamports`

This instruction moves unstaked lamports between accounts with the same authorities and lockups, using the staker authority.

Input structure

```
pub enum StakeInstruction {
    /// # Account references
    ///   0. `[WRITE]` Active or inactive source stake account
```

```
    ///   1. `[WRITE]` Mergeable destination stake account
    ///   2. `[SIGNER]` Stake authority
    ///
    /// The u64 is the portion of available lamports to move
    MoveLamports(u64),
}

fn process_move_lamports(accounts: &[AccountInfo], lamports: u64) ->
    ProgramResult {
```

Parameters

- `lamports`: The amount of unstaked lamports to move.

Accounts

- **`source_stake_account`**: The source stake account to move unstaked lamports from.
    - Signer: No.
    - Init: No.
    - PDA: No.
    - Writable: Yes.
    - Rent checks: None.
    - Ownership checks: The account must be owned by the program.
    - Address checks: The address must be different from the destination stake account.
- **`destination_stake_account`**: The destination stake account to move unstaked lamports to.
    - Signer: No.
    - Init: No.
    - PDA: No.
    - Writable: Yes.
    - Rent checks: None.
    - Ownership checks: The account must be owned by the program.
    - Address checks: The address must be different from the source stake account.
- **`stake_authority`**: The stake authority account who is authorized to manage the stake account.
    - Signer: Yes.
    - Init: No.
    - PDA: No.

DRAFT

- Writable: No.
- Rent checks: None.
- Ownership checks: None.
- Address checks: None.

Additional checks and behavior

- The withdrawal amount must not be equal to zero.
- The stake accounts must be in the `Stake` or `Initialized` state.
- The stake accounts must not be in a transient state, such as activating or deactivating with nonzero effective stake.
- The staker of the source stake account must be the signer.
- The lockups of the source and destination stake accounts must match if they are in force.
- The authorities of the source and destination stake accounts must match.
- The free lamports of the source stake account must be greater than or equal to the amount to move.
- The lamports are moved from the source stake account to the destination stake account.

Instruction: `MoveStake`

This instruction moves stake between accounts with the same authorities and lockups, using the staker authority.

Input structure

```
pub enum StakeInstruction {
    /// # Account references
    ///   0. `[WRITE]` Active source stake account
    ///   1. `[WRITE]` Active or inactive destination stake account
    ///   2. `[SIGNER]` Stake authority
    ///
    /// The u64 is the portion of the stake to move, which may be the entire
    delegation
    MoveStake(u64),
}

fn process_move_stake(accounts: &[AccountInfo], lamports: u64) ->
    ProgramResult {
```

Parameters

- `lamports`: The amount of stake to move.

Accounts

- **`source_stake_account`**: The source stake account to move stake from.
  - Signer: No.
  - Init: No.
  - PDA: No.
  - Writable: Yes.
  - Rent checks: None.
  - Ownership checks: The account must be owned by the program.
  - Address checks: The address must be different from the destination stake account.
- **`destination_stake_account`**: The destination stake account to move stake to.
  - Signer: No.
  - Init: No.
  - PDA: No.
  - Writable: Yes.
  - Rent checks: None.
  - Ownership checks: The account must be owned by the program.
  - Address checks: The address must be different from the source stake account.
- **`stake_authority`**: The stake authority account who is authorized to manage the stake account.
  - Signer: Yes.
  - Init: No.
  - PDA: No.
  - Writable: No.
  - Rent checks: None.
  - Ownership checks: None.
  - Address checks: None.

Additional checks and behavior

- The withdrawal amount must not be equal to zero.
- The stake accounts must be in the `Stake` or `Initialized` state.
- The stake accounts must not be in a transient state, such as activating or deactivating with nonzero effective stake.

DRAFT

- The staker of the source stake account must be the signer.
- The lockups of the source and destination stake accounts must match if they are in force.
- The authorities of the source and destination stake accounts must match.
- The size of `source_stake_account` and `destination_stake_account` must be equal to the size of `StakeStateV2`.
- The destination merge kind must be fully active or inactive.
- The source stake account must have enough stake to move.
- The source stake account must retain at least the minimum delegation if not all stake is being moved.
- If the destination stake account is active, the destination stake account must retain at least the minimum delegation.
- If the destination stake account is inactive, lamports to move must not be less than the minimum delegation.
- Stake and lamports are moved from the source stake account to the destination stake account.
- An error is returned if the source or destination account balance is less than the rent-exempt reserve.

Instruction: `Redelegate`

This instruction is deprecated. The program will return an error if this instruction is called.

Instruction: `SetLockup`

This instruction updates the lockup information of a stake account. If the lockup is active, the lockup custodian may update the lockup parameters. If the lockup is not active, the withdraw authority may set a new lockup.

Input structure

```
pub struct LockupArgs {
    pub unix_timestamp: Option<UnixTimestamp>,
    pub epoch: Option<Epoch>,
    pub custodian: Option<Pubkey>,
}

pub enum StakeInstruction {
    /// # Account references
    ///   0. `[WRITE]` Initialized stake account
    ///   1. `[SIGNER]` Lockup authority or withdraw authority
    SetLockup(LockupArgs),
```

```
}
```

Parameters

- `unix_timestamp`: Unix timestamp when the lockup expires.
- `epoch`: Epoch when the lockup expires.
- `custodian`: Custodian key on a transaction exempts the operation from lockup constraints.

Accounts

- **`stake_account`**: The initialized stake account whose lockup information will be updated.
  - Signer: No.
  - Init: No.
  - PDA: No.
  - Writable: Yes.
  - Rent checks: None.
  - Ownership checks: The account must be owned by the program.
  - Address checks: None.
- **`lockup_or_withdraw_authority`**: The lockup authority or withdraw authority account must be a signer.
  - Signer: Yes.
  - Init: No.
  - PDA: No.
  - Writable: No.
  - Rent checks: None.
  - Ownership checks: None.
  - Address checks: None.

Additional checks and behavior

- Only the lockup custodian can update the lockup while it is in force.
- Only the withdraw authority can set a new lockup if the lockup is not in force.

Instruction: `SetLockupChecked`

This instruction updates the lockup information of a stake account. if the lockup is active, the lockup custodian may update the lockup parameters. If the lockup is not active, the withdraw

DRAFT

authority may set a new lockup. This instruction behaves like `SetLockup` with the additional requirement that the new lockup authority also be a signer.

Input structure

```
pub struct LockupCheckedArgs {
    pub unix_timestamp: Option<UnixTimestamp>,
    pub epoch: Option<Epoch>,
}

pub enum StakeInstruction {
    /// # Account references
    ///   0. `[WRITE]` Initialized stake account
    ///   1. `[SIGNER]` Lockup authority or withdraw authority
    ///   2. Optional: `[SIGNER]` New lockup authority
    SetLockupChecked(LockupCheckedArgs),
}
```

Parameters

- `unix_timestamp`: Unix timestamp when the lockup expires.
- `epoch`: Epoch when the lockup expires.

Accounts

- **`stake_account`**: The initialized stake account whose lockup information will be updated.
  - Signer: No.
  - Init: No.
  - PDA: No.
  - Writable: Yes.
  - Rent checks: None.
  - Ownership checks: The account must be owned by the program.
  - Address checks: None.
- **`old_withdraw_or_lockup_authority`**: Unused.
- **`option_new_lockup_authority`** (optional): The new lockup authority account to be a custodian.
  - Signer: Yes.
  - Init: No.
  - PDA: No.

DRAFT

- Writable: No.
- Rent checks: None.
- Ownership checks: None.
- Address checks: None.

Additional checks and behavior

- Only the lockup custodian can update the lockup while it is in force.
- Only the withdraw authority can set a new lockup if the lockup is not in force.
- The new lockup authority must be a signer.

Instruction: `Split`

This instruction splits tokens and stake off a stake account into a new stake account.

Input structure

```
pub enum StakeInstruction {
    /// # Account references
    ///   0. `[WRITE]` Stake account to be split; must be in the Initialized or
    Stake state
    ///   1. `[WRITE]` Uninitialized stake account that will take the
    split-off amount
    ///   2. `[SIGNER]` Stake authority
    Split(u64),
}

fn process_split(accounts: &[AccountInfo], split_lamports: u64) ->
    ProgramResult {
```

Parameters

- `split_lamports`: The amount of lamports to split off from the source stake account.

Accounts

- **`source_stake_account`**: The initialized or delegated stake account to be split.
  - Signer: No.
  - Init: No.

    - PDA: No.
    - Writable: Yes.
    - Rent checks: The account must be rent-exempt.
    - Ownership checks: The account must be owned by the program.
    - Address checks: None.
- **`destination_stake_account`**: The uninitialized stake account that will take the split-off amount.
    - Signer: No.
    - Init: No.
    - PDA: No.
    - Writable: Yes.
    - Rent checks: None.
    - Ownership checks: The account must be owned by the Solana vote program.
    - Address checks: None.
- **`stake_authority`**: The stake authority account who is authorized to manage the stake account.
    - Signer: Yes.
    - Init: No.
    - PDA: No.
    - Writable: No.
    - Rent checks: None.
    - Ownership checks: None.
    - Address checks: None.

Additional checks and behavior

- The size of the destination stake account must be equal to the size of `StakeStateV2`.
- The destination stake account must be uninitialized.
- The lamports to be split must be less than or equal to the lamports in the source stake account.
- The staker of the source stake account must be a signer.
- The lamports to be split must be greater than zero and less than or equal to the total lamports in the source stake account.
- The remaining balance of the source stake account after the split must be at least the rent exempt reserve plus the additional required lamports, which can be zero or the minimum delegation amount.
- If the source stake account is active, which means it has a effective stake, then one of the following criteria must be met: 1) the destination stake account must be prefunded with at least the rent-exempt reserve, or 2) the split must consume 100% of the source stake account.

DRAFT

- The split amount must be less than the rent reserve plus the additional required lamports minus the current destination balance.
- If the source stake account is in the `Stake` state, the following takes place:
    - If the remaining balance of the source stake account is zero, the remaining stake delta and split stake amount must be equal. Otherwise, the stake amount of the source stake account minus the split amount must be greater than or equal to the minimum delegation amount.
    - The split stake amount must be greater than or equal to the minimum delegation amount.
    - The source stake account is split into the destination stake account.
- If the source stake account is uninitialized, the source stake account must be a signer.
- Copy the source stake account meta to the destination stake account meta with new rent-exempt reserve.
- The source stake account is deinitialized if the split amount is equal to the total lamports in the source stake account.
- The split amount of lamports is moved from the source stake account to the destination stake account.

Instruction: `Withdraw`

This instruction withdraws unstaked lamports from a stake account.

Input structure

```
pub enum StakeInstruction {
    /// # Account references
    ///   0. `[WRITE]` Stake account from which to withdraw
    ///   1. `[WRITE]` Recipient account
    ///   2. `[]` Clock sysvar
    ///   3. `[]` Stake history sysvar that carries stake warmup/cooldown
    history
    ///   4. `[SIGNER]` Withdraw authority
    ///   5. Optional: `[SIGNER]` Lockup authority, if before lockup expiration
    ///
    /// The u64 is the portion of the stake account balance to be withdrawn,
    ///    must be `<= StakeAccount.lamports - staked_lamports`.
    Withdraw(u64),
}

fn process_withdraw(accounts: &[AccountInfo], withdraw_lamports: u64) ->
    ProgramResult {
```

Parameters

- `withdraw_lamports`: The amount of unstaked lamports to withdraw.

Accounts

- **`source_stake_account`**: The stake account from which to withdraw.
  - Signer: No.
  - Init: No.
  - PDA: No.
  - Writable: Yes.
  - Rent checks: None.
  - Ownership checks: The account must be owned by the program.
  - Address checks: None.
- **`destination`**: The recipient account.
  - Signer: No.
  - Init: No.
  - PDA: No.
  - Writable: Yes.
  - Rent checks: None.
  - Ownership checks: None — but only lamports' balance is updated.
  - Address checks: None.
- **`clock`**: The account must be a clock sysvar.
- **`stake_history`**: Unused.
- **`withdraw_authority`**: The withdraw authority account who is authorized to withdraw from the stake account.
  - Signer: Yes.
  - Init: No.
  - PDA: No.
  - Writable: No.
  - Rent checks: None.
  - Ownership checks: None.
  - Address checks: None.
- **`option_lockup_authority`** (optional): The lockup authority account if before lockup expiration.
  - Signer: Yes.
  - Init: No.
  - PDA: No.

DRAFT

- Writable: No.
- Rent checks: None.
- Ownership checks: None.
- Address checks: None.

Additional checks and behavior

- `withdraw_authority` must be a signer.
- `option_lockup_authority` must be a signer if provided.
- Signers must contain the withdraw authority and optionally the lockup authority if provided.
- The stake account must have sufficient lamports to cover the withdrawal amount plus any required reserve.
- If the stake account is in the `Stake` state, and if the current epoch is greater than or equal to the deactivation epoch, the stake amount is calculated using the stake history and cooldown. Otherwise, the stake amount is the delegation stake.
- If the stake account is in the `Uninitialized` state, the `source_stake_account` must be a signer.
- The lockup must have expired or the custodian must be a signer if the lockup is in force.
- If the withdrawal amount is equal to the stake account balance and the account is still active, the withdrawal is rejected.
- If the withdrawal amount is equal to the stake account balance, the account is deinitialized.
- If the withdrawal amount is less than the stake account balance, the withdrawal amount must not deplete the reserve.

DRAFT

5. Assessment Results

At the time of our assessment, the reviewed code was not deployed on Solana Mainnet.

During our assessment on the scoped BPF Stake Program programs, there were no security vulnerabilities discovered.

5.1. Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the version reviewed during our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.

DRAFT