Zellic

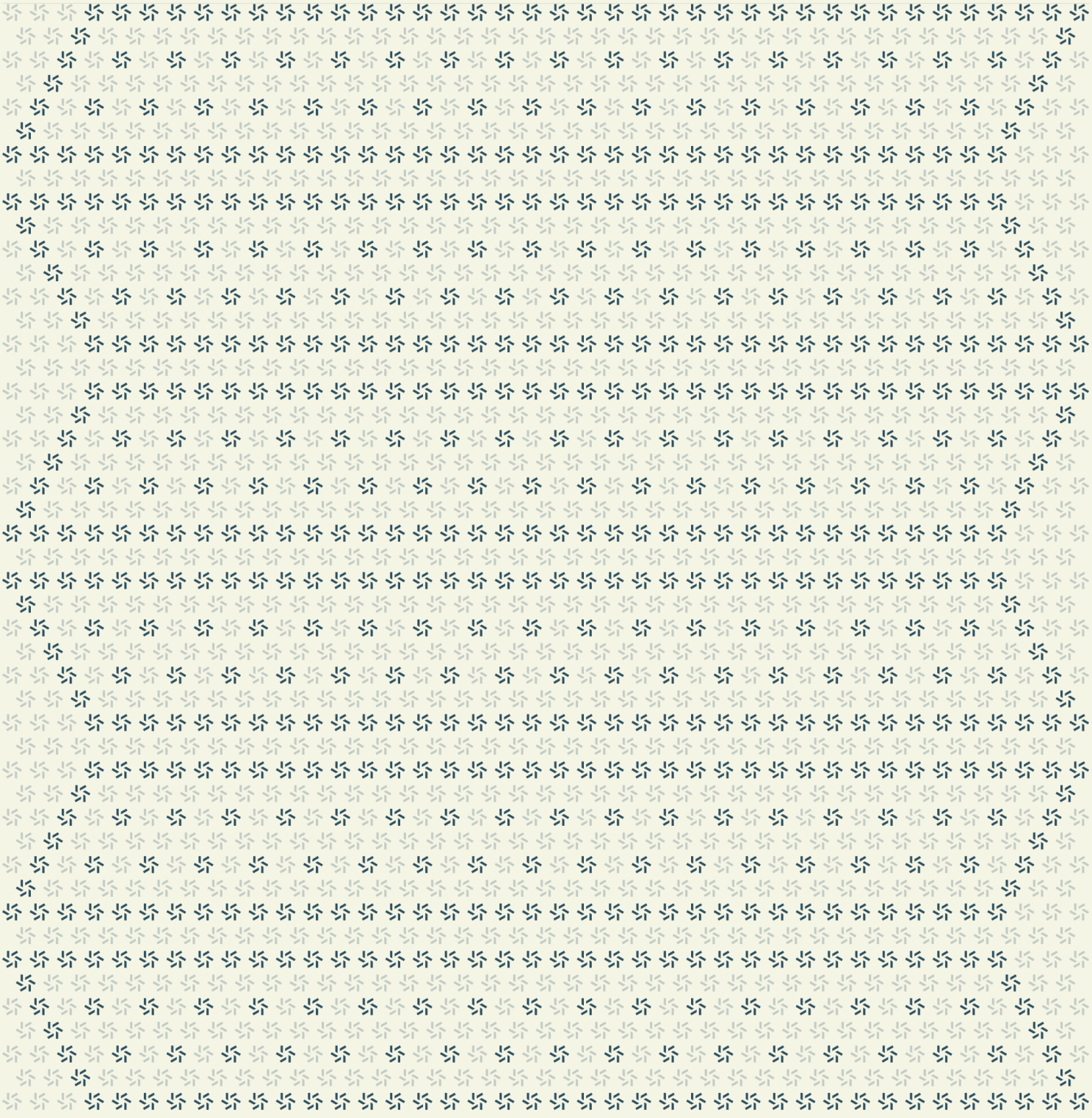
Prepared for
Karolina
Blueprint Finance

Prepared by
Frank Bachman
Bryce Casaje
Zellic

August 13, 2025

Glow Protocol

Solana Application Security Assessment



Contents

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 ↗.

1. Overview

1.1. Executive Summary

Zellic conducted a security assessment for Blueprint Finance from June 27th to July 21st, 2025. During this engagement, Zellic reviewed Glow Protocol'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 there logic bugs that could result in users performing actions that they are not permitted to?
- Are there valuation errors with the Pyth oracle implementation?
- Are users able to register positions that bypass `token_features` isolation?

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 Glow Protocol programs, we discovered 10 findings. No critical issues were found. Five findings were of high impact, four were of medium impact, and one was of low impact.

Additionally, Zellic recorded its notes and observations from the assessment for the benefit of Blueprint Finance in the Discussion section (4. ↗).

Based on the number of severe findings uncovered during the audit, it is our opinion that the project is not yet ready for production. We strongly advise a comprehensive reassessment before

deployment to help identify any potential issues or vulnerabilities introduced by necessary fixes or changes. We also recommend adopting a security-focused development workflow, including (but not limited to) augmenting the repository with comprehensive end-to-end tests that achieve 100% branch coverage using any common, maintainable testing framework, thoroughly documenting all function requirements, and training developers to have a security mindset while writing code.

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	5
Medium	4
Low	1
Informational	0

2. Introduction

2.1. About Glow Protocol

Blueprint Finance contributed the following description of Glow Protocol:

> Glow Protocol is a leveraged protocol that allows users leverage across supported integrations in the Solana ecosystem. It works though adapters that integrate with other programs, allowing users to transact with leverage. The protocol currently includes borrowing & lending and leveraged staking using Glow's GlowSOL.

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. 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 (4. ↗) 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.

2.3. Scope

The engagement involved a review of the following targets:

Glow Protocol Programs

Type	Rust
Platform	Solana
Target	glow-v1
Repository	https://github.com/Blueprint-Finance/glow-v1 ↗
Version	`1a2ac73cccfd316c26703ada8724f6155b73df72`

Programs	`programs/margin/**`
	`programs/margin-pool/**`
	`programs/airspace/**`
	`programs/metadata/**`
	`libraries/rust/program-common/**`
	`libraries/rust/program-proc-macros/**`

Target	lookup-table-registry
Repository	https://github.com/Blueprint-Finance/lookup-table-registry ↗
Version	`71ce34e338b0266dcba5ea351e052531707004c9`
Programs	`programs/**`

2.4. Project Overview

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

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

Pedro Moura
Engagement Manager
pedro@zellic.io

The following consultants were engaged to conduct the assessment:

Frank Bachman
Engineer
frank@zellic.io

Bryce Casaje
Engineer
bryce@zellic.io

2.5. Project Timeline

The key dates of the engagement are detailed below.

June 27, 2025	Start of primary review period
June 30, 2025	Kick-off call
July 21, 2025	End of primary review period

3. Detailed Findings

3.1. Restricted airspace permit has incorrect issuer ID

Target	airspace		
Category	Business Logic	**Severity**	High
Likelihood	High	**Impact**	High

Description

When a permit is issued by the airspace's authority, the permit issuer is set to the airspace's key instead of the actual issuer's key.

```
permit.issuer = if airspace.authority == authority.key() {
    airspace.key()
} else {
    authority.key()
};
```

This causes a flaw in the revocation logic for restricted airspaces.

The `airspace_permit_revoke` instruction attempts to load an `AirspacePermitIssuerId` account using the permit's issuer as a seed:

```
#[account(
    seeds = [
        AIRSPACE_PERMIT_ISSUER,
        airspace.key().as_ref(),
        permit.issuer.as_ref()
    ],
    bump
)]
issuer_id: AccountInfo<'info>,
```

The revocation logic checks three conditions to determine if revocation should be blocked:

```
// programs/airspace/src/instructions/airspace_permit_revoke.rs:69-85
// The airspace authority, who can always revoke
let is_airspace_authority = authority == airspace.authority;
// The permit issuer, who can always revoke
let is_airspace_permit_issuer = authority == permit.issuer;
// Check if the airspace is restricted and the regulator license has NOT been
```

```
    revoked,
// in which case only the above can revoke the permit.
let is_restricted_issuer_not_revoked =
    airspace.is_restricted && !ctx.accounts.issuer_id.data_is_empty();

// If the signer is:
//   * NOT the airspace authority
//   * NOT the permit issuer
//   * anyone else, in a restricted airspace, and the permit issuer is NOT
    revoked
// then the permit can NOT be revoked.
if !is_airspace_authority && !is_airspace_permit_issuer &&
    is_restricted_issuer_not_revoked {
    return err!(AirspaceErrorCode::PermissionDenied);
}
```

Since no `AirspacePermitIssuerId` account exists for the airspace itself, `issuer_id.data_is_empty()` always returns `true`, making `is_restricted_issuer_not_revoked = false`.

Since `is_restricted_issuer_not_revoked` is false, the check to deny revocation permission fails, allowing anyone to revoke the permit.

Impact

This vulnerability breaks the access-control mechanism for permit revocation.

This could lead to the following:

- **Denial-of-service attacks.** Malicious actors can revoke legitimate permits, preventing authorized users from accessing airspace resources.
- **Economic disruption.** Users may lose access to positions or assets tied to their permits.

This has a similar impact to Finding 3.3. ↗.

Recommendations

Fix the permit-creation logic to either not set the issuer ID or to correctly load the `AirspacePermitIssuerId`.

Remediation

This issue has been acknowledged by Blueprint Finance, and a fix was implemented in commit `00b77f92` ↗.

3.2. Instruction `collect_liquidation_fee` uses a local copy of `Liquidation-State`

Target	margin		
Category	Coding Mistakes	**Severity**	High
Likelihood	High	**Impact**	High

Description

The `LiquidationState` account is defined with `#[account(zero_copy)]` to allow for direct memory operations. However, the `Liquidation` struct nested inside it also derives the `Copy` trait.

In the `collect_liquidation_fee` instruction, the state is loaded like this:

```
let mut liquidation = liquidation_state.state;
```

When accessing the `.state` field by value, the compiler creates a local copy of the `Liquidation` data instead of returning a mutable reference into the `zero_copy` account buffer.

This means any modifications to `liquidation` (like setting `is_collecting_fees` or clearing fee slots) are made to a temporary, in-memory copy and are not persisted back to the on-chain account data.

Impact

This vulnerability causes the `collect_liquidation_fee` instruction to fail to update the `LiquidationState` account correctly. The `is_collecting_fees` flag is never set, and the fee slots are not cleared. This can lead to incorrect fee collection and potentially allow liquidators to collect fees multiple times for the same liquidation.

Recommendations

The `collect_liquidation_fee` instruction should be updated to use a mutable reference to the `LiquidationState` account's state. This can be done by changing the line

```
let mut liquidation = ctx.accounts.liquidation.load_mut()?.state;
```

to the following:

```
let liquidation = &mut ctx.accounts.liquidation.load_mut()?.state;
```

This will ensure that all modifications to the `liquidation` variable are made directly to the on-chain account data.

Remediation

This issue has been acknowledged by Blueprint Finance, and a fix was implemented in commit `7bd8f87a` ↗.

3.3. Anyone can revoke a permit in an unrestricted airspace

Target	airspace		
Category	Business Logic	**Severity**	High
Likelihood	High	**Impact**	High

Description

In the `airspace_permit_revoke` instruction, the logic to check if a permit can be revoked is flawed for unrestricted airspaces.

The check `is_restricted_issuer_not_revoked` is intended to allow anyone to revoke a permit in a restricted airspace if the issuer's license has been revoked. However, for unrestricted airspaces, `airspace.is_restricted` is `false`, which causes `is_restricted_issuer_not_revoked` to always be `false`.

This allows the if condition that prevents unauthorized revocation to be bypassed, meaning anyone can revoke a permit for an unrestricted airspace.

```
pub fn airspace_permit_revoke_handler(ctx: Context<AirspacePermitRevoke>) ->
    Result<()> {
    let airspace = &mut ctx.accounts.airspace;
    let permit = &ctx.accounts.permit;
    let authority = ctx.accounts.authority.key();

    // The airspace authority, who can always revoke
    let is_airspace_authority = authority == airspace.authority;
    // The permit issuer, who can always revoke
    let is_airspace_permit_issuer = authority == permit.issuer;
    // Check if the airspace is restricted and the regulator license has NOT
    been revoked,
    // in which case only the above can revoke the permit.
    let is_restricted_issuer_not_revoked =
        airspace.is_restricted && !ctx.accounts.issuer_id.data_is_empty();

    // If the signer is:
    //   * NOT the airspace authority
    //   * NOT the permit issuer
    //   * anyone else, in a restricted airspace, and the permit issuer is NOT
    revoked
    // then the permit can NOT be revoked.
    if !is_airspace_authority && !is_airspace_permit_issuer &&
```

```
    is_restricted_issuer_not_revoked {
        return err!(AirspaceErrorCode::PermissionDenied);
    }

    emit!(AirspacePermitRevoked {
        airspace: airspace.key(),
        permit: permit.key(),
    });

    Ok(())
}
```

Impact

This vulnerability allows any user to revoke any permit in an unrestricted airspace, potentially disrupting the operations of the protocol and its users.

This has a similar impact to Finding 3.1. ↗.

Recommendations

The logic in `airspace_permit_revoke_handler` should be updated to correctly handle unrestricted airspaces. A specific check should be added to ensure that for unrestricted airspaces, only the airspace authority or the permit issuer can revoke a permit.

Remediation

This issue has been acknowledged by Blueprint Finance, and a fix was implemented in commit `c216ef54` ↗.

3.4. Governor can transfer user positions and loans

Target	margin, margin-pool		
Category	Business Logic	**Severity**	High
Likelihood	Low	**Impact**	High

Description

Two administrative instructions allow the protocol governor to transfer user assets without consent, creating significant centralization risks:

1. The first, **`admin_transfer_position` (margin program)**, allows the governor to transfer any amount of a user's position to another margin account.
2. The other, **`admin_transfer_loan` (margin-pool program)**, allows the governor to transfer debt obligations (loans) from one user to another.

```
// admin_transfer_position (margin program)
#[derive(Accounts)]
pub struct AdminTransferPosition<'info> {
    /// The administrative authority
    #[account(address = PROTOCOL_GOVERNOR_ID)]
    pub authority: Signer<'info>,

    /// The target margin account to move a position into
    #[account(mut)]
    pub target_account: AccountLoader<'info, MarginAccount>,

    /// The source account to move a position out of
    #[account(mut)]
    pub source_account: AccountLoader<'info, MarginAccount>,
    // ...
}

// admin_transfer_loan (margin-pool program)
#[derive(Accounts)]
pub struct AdminTransferLoan<'info> {
    /// The administrative authority
    #[account(address = PROTOCOL_GOVERNOR_ID)]
    pub authority: Signer<'info>,

    /// The loan account to be moved from
```

```
    #[account(mut, token::authority = margin_pool)]
    pub source_loan_account: InterfaceAccount<'info, TokenAccount>,

    /// The loan account to be moved into
    #[account(mut, token::authority = margin_pool)]
    pub target_loan_account: InterfaceAccount<'info, TokenAccount>,
    // ...
}
```

Both instructions have similar security issues:

- The `admin_transfer_position` instruction does not check if source and target margin accounts belong to the same airspace, allowing transfers that circumvent airspace isolation guarantees.
- Neither instruction requires consent from the affected users.

Impact

A compromised governor could use these instructions to do the following:

1. **Drain user funds.** They could transfer positions or assign debt obligations to arbitrary accounts, effectively stealing from users.
2. **Circumvent security controls.** They could move assets between different airspaces, bypassing isolation guarantees and potentially moving assets from restricted to unrestricted airspaces.
3. **Manipulate debt obligations.** They could arbitrarily assign loans to users who never borrowed funds.

Recommendations

Both administrative transfer instructions should be removed or significantly restricted. If deemed necessary for administrative purposes, they should be subject to timelock mechanisms, multi-signature approval, clear usage guidelines that are communicated to users, and checks to prevent transfers between airspaces.

Remediation

The Blueprint Finance team acknowledged the issue, and provided the following comment:

> Our intention is to use the instructions in the worst case where a margin account's equity becomes negative. A liquidator would liquidate all assets to repay debts, and leave a margin ac-

count only having debts. Outside of those admin functions, we currently don't have a mechanism to socialise losses. [We are working on this for a future release]

We also use a multisig to manage our protocol governance, we consider that to be a mitigation too, and we have internal controls to ensure that not only the developer team sign and execute transactions, but management are also involved.

3.5. Two-step finalize mechanism breaks on incorrect pubkey

Target	airspace		
Category	Business Logic	**Severity**	High
Likelihood	Low	**Impact**	High

Description

The two-step propose-then-finalize mechanism for transferring authority of an airspace or governor breaks if a transfer is proposed to an incorrect public key. The `AuthorityTransfer` struct, created during the propose step, can only be closed by a successful finalize instruction, which requires a signature from the proposed new authority:

```
#[derive(Accounts)]
pub struct AirspaceAuthorityFinalize<'info> {
    proposed_authority: Signer<'info>,

    #[account(
        mut,
        close = old_authority,
        constraint = transfer.resource == airspace.key(),
        constraint = transfer.current_authority == old_authority.key(),
        constraint = transfer.new_authority == proposed_authority.key(),
    )]
    transfer: Account<'info, AuthorityTransfer>,

    // ...
}
```

If this proposed new authority is an incorrect pubkey, the finalize transaction can never be signed, which prevents the transfer from completing. The `airspace_propose_authority` and `governor_propose` instructions create `AuthorityTransfer` accounts at a PDA derived from the airspace or governor ID, restricting the transfer mechanism such that only one transfer can be pending at a time:

```
#[derive(Accounts)]
#[instruction(proposed_governor: Pubkey)]
pub struct GovernorPropose<'info> {
    // ...

    /// The choice of seeds ensures that there can only ever be 1 transfer
```

```
    pending
    #[account(
        init,
        seeds = [
            GOVERNOR_ID,
            governor_id.key().as_ref(),
        ],
        bump,
        space = 8 + std::mem::size_of::<AuthorityTransfer>(),
        payer = governor,
    )]
    transfer: Account<'info, AuthorityTransfer>,

    // ...
}
```

Since the finalize instruction requires a signature from the `proposed_authority`, if the `proposed_authority` is an incorrect public key, the finalize transaction can never be successfully executed. This permanently locks the authority-transfer mechanism, as a new `AuthorityTransfer` cannot be initiated while the previous one is still pending.

Impact

This vulnerability can lead to a permanent loss of control over an airspace or the entire protocol's governance. If an authority transfer is proposed to an incorrect public key, the ability to transfer the authority in the future is lost.

Recommendations

A mechanism to cancel a pending authority transfer should be implemented, allowing for the correction of mistakes and preventing the permanent locking of the authority-transfer mechanism.

Remediation

This issue has been acknowledged by Blueprint Finance, and a fix was implemented in commit 20ce6d07 ↗.

3.6. Margin register-position inconsistency

Target	margin		
Category	Business Logic	**Severity**	Medium
Likelihood	High	**Impact**	Medium

Description

The `register_position` instruction in the margin program is intended to register adapter-administered tokens (`TokenAdmin::Adapter`). However, the implementation has inconsistent validation logic that prevents proper separation between token types. The system allows configuring tokens with mismatched admin types and kinds, leading to tokens that cannot be properly registered through the intended instruction.

```
account.register_position(
    PositionConfigUpdate::new_from_config(
        config,
        position_token.decimals,
        address,
        config
            .adapter_program()
            .ok_or_else(|| error!(ErrorCode::InvalidConfigRegisterPosition))?,
    )?,
    &[Approver::MarginAccountAuthority],
)?;
```

The call to `account.register_position` in `register_position` sets the adapter argument as `config.adapter_program().ok_or_else(|| error!(ErrorCode::InvalidConfigRegisterPosition))?`. This means that adapter-administered tokens (`TokenAdmin::Adapter`) can be registered, but margin-administered tokens (`TokenAdmin::Margin`) will fail immediately.

Then, for adapter tokens that pass the first check, `MarginAccount::register_position` calls `AccountPosition::may_be_registered_or_closed` for a newly added position. The hardcoded call only contains `[Approver::MarginAccountAuthority]`, so `authority_approved` will always be `true`, and `adapter_approved` will always be `false`.

```
pub fn may_be_registered_or_closed(&self, approvals: &[Approver]) -> bool {
    let mut authority_approved = false;
    let mut adapter_approved = false;
    for approval in approvals {
```

```
            match approval {
                Approver::MarginAccountAuthority => authority_approved = true,
                Approver::Adapter(approving_adapter) => {
                    adapter_approved = *approving_adapter == self.adapter
                }
            }
        }
        match self.kind() {
            TokenKind::Collateral => authority_approved && !adapter_approved,
            TokenKind::Claim | TokenKind::AdapterCollateral => {
                authority_approved && adapter_approved
            }
        }
    }
```

This means that the `register_position` instruction appears to only work for adapter tokens that are of kind `TokenKind::Collateral`. All other token types fail either the adapter requirement or approval checks.

Impact

This inconsistency can lead to confusion and operational issues for developers and users. The mismatched validation logic between token configuration and registration creates a system where certain token configurations cannot be used through their intended registration paths, potentially leading to developer errors and broken integrations.

Recommendations

The `register_position` instruction should be updated to strictly enforce that only adapter-administered tokens can be registered. The `configure_token` instruction should also be reviewed to ensure that it correctly validates the token admin and kind, preventing the creation of inconsistent token configurations.

Remediation

This issue has been acknowledged by Blueprint Finance, and a fix was implemented in commit `96fe45b9` ↗.

3.7. Registry state corruption when reusing empty slots

Target	lookup-table-registry		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	High	**Impact**	Medium

Description

The `create_lookup_table` instruction contains a bug in its state-management logic that corrupts the registry's `len` and `capacity` counters when reusing previously closed slots. This leads to permanent registry breakage and fund loss.

The bug occurs in the slot reuse path:

```
let (len, capacity) = {
    let registry = &ctx.accounts.registry_account;
    (registry.len, registry.capacity)
};
let append_to_end = len == capacity;

if append_to_end {
    ctx.accounts.registry_account.len += 1;
}

// ...
if append_to_end {
    // ...
} else {
    let slot = ctx.accounts.registry_account.find_empty_entry()?;
    *slot = entry;
}

ctx.accounts.registry_account.capacity += 1;
```

When reusing an empty slot, the instruction fails to increment `len` but still increments `capacity`, causing the registry's state counters to become permanently desynchronized from the actual number of active entries.

Impact

If this bug occurs, the registry would not be able to be closed because the `close_registry_account` instruction requires `len == 0`, but `len` becomes understated relative to actual entries.

In addition, subsequent `remove_lookup_table` operations would panic when they attempt `len.checked_sub(1).unwrap()` on an invalid `len` value, making the registry completely unusable.

Recommendations

Fix the state-update logic to correctly handle both allocation and reuse cases:

```
if append_to_end {
    // Allocating new slot
    ctx.accounts.registry_account.len += 1;
    ctx.accounts.registry_account.capacity += 1;
} else {
    // Reusing empty slot
    ctx.accounts.registry_account.len += 1;
    // capacity should NOT be incremented when reusing
    let slot = ctx.accounts.registry_account.find_empty_entry()?;
    *slot = entry;
}
```

Additionally, add comprehensive tests covering remove/re-add cycles to prevent similar state-management bugs.

Remediation

This issue has been acknowledged by Blueprint Finance, and a fix was implemented in commit e7af16f4 ↗.

3.8. Registry-creation denial of service via rent manipulation

Target	lookup-table-registry		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	High	**Impact**	Medium

Description

The `create_lookup_table` instruction contains a denial-of-service vulnerability in its rent-calculation logic. When computing the lamport transfer amount needed to resize the registry account, the code uses `checked_sub().unwrap()`, which panics if the account already has sufficient lamports.

```
let transfer_amount = rent
    .minimum_balance(new_size)
    .checked_sub(registry_info.lamports())
    .unwrap();
```

If `registry_info.lamports()` is greater than or equal to the required minimum balance, `checked_sub()` returns `None` and `unwrap()` panics, causing the entire transaction to revert.

Impact

This vulnerability enables a trivial denial-of-service attack if any external actor sends lamports to the registry PDA address.

By prefunding the registry account to equal or exceed the required balance for the enlarged account size, an attacker can cause all subsequent `create_lookup_table` calls to panic and fail.

The attack only costs transaction fees and a small amount of lamports sent to the target registry and persists until someone drains the excess lamports from the registry account.

Recommendations

Replace the unsafe arithmetic with proper overflow handling:

```
let required = rent.minimum_balance(new_size);
let transfer_amount = required.saturating_sub(registry_info.lamports());

if transfer_amount > 0 {
```

```
    // Perform the transfer
}
```

Alternatively, use `unwrap_or`.

```
let transfer_amount = rent
    .minimum_balance(new_size)
    .checked_sub(registry_info.lamports())
    .unwrap_or(0);
```

Remediation

This issue has been acknowledged by Blueprint Finance, and a fix was implemented in commit 398ab566 ↗.

3.9. Token-config reinitialization bypasses update checks

Target	margin		
Category	Business Logic	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

The `configure_token` instruction, designed to manage `TokenConfig` accounts, utilizes Anchor's `init_if_needed` attribute. This allows the same instruction to either initialize a new `TokenConfig` account or update an existing one. The program distinguishes between these two operations using an `is_init` flag, which is set to `true` if `config.mint` is `Pubkey::default()` (indicating a newly created account).

The `configure_token` instruction can be used to close an existing `TokenConfig` account by passing `None` as `updated_config`:

```
pub fn configure_token_handler(
    ctx: Context<ConfigureToken>,
    updated_config: Option<TokenConfigUpdate>,
) -> Result<()> {
    let config = &mut ctx.accounts.token_config;

    // ...

    let updated_config = match updated_config {
        Some(update) => update,
        None => return config.close(ctx.accounts.payer.to_account_info()),
    };

    // ...
}
```

An issue arises because several vital immutability checks are only enforced when `is_init` is `false`, meaning they apply exclusively to updates of preexisting configurations. If an attacker uses the `configure_token` instruction to close the `TokenConfig` account, they can then recreate it using the same seeds via a subsequent call to the same instruction.

This recreation is treated as an initialization by the program (`is_init` becomes `true`), allowing the attacker to bypass all checks that would normally prevent changes to critical token properties.

These bypassed checks include the following:

- Preventing changes to `token_kind`
- Enforcing `admin` type transitions (e.g., disallowing the switch from `Adapter` to `Margin`)
- Maintaining `adapter` address immutability
- Restricting changes to `token_features` (excluding the `RESTRICTED` flag, which is designed to be toggleable)

Impact

This vulnerability allows an attacker to bypass intended immutability constraints on `TokenConfig` accounts.

This could lead to protocol inconsistency or exploitation of other program logic that expects certain token config fields to stay constant.

Recommendations

Robust checks should be implemented to prevent reinitialization from bypassing update constraints.

For example, consider removing `init_if_needed` and having separate `init_token_config` and `update_token_config` instructions, each with appropriate checks.

Alternatively, prevent the closure of `TokenConfig` accounts in general if they have critical immutable properties.

Remediation

This issue has been acknowledged by Blueprint Finance, and a fix was implemented in commit `b7faccbe` ↗.

3.10. Instruction `configure_token` should validate `underlying_mint_token_program`

Target	margin		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The `configure_token` instruction does not validate that the `underlying_mint_token_program` field in the `TokenConfigUpdate` is a valid token program. An administrator could inadvertently set this to an arbitrary public key.

```
config.mint = ctx.accounts.mint.key();
config.mint_token_program = *ctx.accounts.mint.to_account_info().owner;
config.airspace = ctx.accounts.airspace.key();
config.underlying_mint = updated_config.underlying_mint;
config.underlying_mint_token_program
    = updated_config.underlying_mint_token_program; // not validated
config.admin = updated_config.admin;
config.token_kind = updated_config.token_kind;
config.value_modifier = updated_config.value_modifier;
config.max_staleness = updated_config.max_staleness;
config.token_features = updated_config.token_features;
```

The misconfigured `underlying_mint_token_program` would be used when creating position configurations via `PositionConfigUpdate::new_from_config()`, which is called in instructions like `register_position`:

```
account.register_position(
    PositionConfigUpdate::new_from_config(
        config,
        position_token.decimals,
        address,
        config
            .adapter_program()
            .ok_or_else(|| error!(ErrorCode::InvalidConfigRegisterPosition))?,
    )?,
    &[Approver::MarginAccountAuthority],
)?;
```

The `new_from_config` function selects the token program based on token kind and validates it:

```
pub fn new_from_config(
    config: &Account<TokenConfig>,
    mint_decimals: u8,
    address: Pubkey,
    adapter: Pubkey,
) -> anchor_lang::Result<Self> {
    let token_program = match config.token_kind {
        TokenKind::Collateral => config.underlying_mint_token_program,
        TokenKind::Claim => config.mint_token_program,
        TokenKind::AdapterCollateral => config.mint_token_program,
    };
    // Check that the token program is a supported token program
    if
    !anchor_spl::token_interface::TokenAccount::owners().contains(&token_program)
    {
        msg!("Unsupported token program: {}", token_program);
        return err!(ErrorCode::UnknownTokenProgram);
    }
    // ...
}
```

Impact

If the `underlying_mint_token_program` is invalid, downstream validation will catch it and cause position registration to fail. This prevents any direct loss of funds but results in a denial of service for the specific token until the configuration is corrected.

Recommendations

The `configure_token` instruction should be updated to validate that the `underlying_mint_token_program` is a valid and known token program address (e.g., `spl-token` or `spl-token-2022`).

Remediation

This issue has been acknowledged by Blueprint Finance, and a fix was implemented in commit `bf57aade` ↗.

4. 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.

4.1. Improvements to account constraints

During the audit, several instructions were identified where account constraints could be strengthened. While these cases do not represent currently exploitable vulnerabilities, adding stricter top-level constraints provides better defense in depth, improves code clarity, and makes the programs more robust against future code changes.

For example, many instructions take an authority account and a margin account and then verify the authority is allowed to modify the margin account by later calling `margin_account.verify_authority(authority.key())?;`. The relationship between the two accounts is not made clear in the account constraints. In addition, `verify_authority` could get moved around in future code changes. Adding a constraint to the account struct would help improve the clarity and reduce security risk for these instructions.

In addition, in `metadata::remove_entry` and `margin::close_account`, the `receiver` account should be constrained as a `SystemAccount` because it receives rent from closed accounts and should be a user-owned wallet. This is not currently a vulnerability because providing an invalid account (e.g., a PDA owned by another program) would only cause the user's own transaction to fail, not affect the protocol or other users.

In instructions like `margin::transfer_deposit`, accounts used as token-transfer authorities (`source_owner`) are passed as `AccountInfo` but should be passed as `Signer`. This is not a vulnerability because the downstream token program correctly validates that the authority is a transaction signer. However, making it an explicit `Signer` constraint improves clarity and security posture.

A similar and more critical example is the `margin::refresh_deposit_position` instruction. This instruction accepts an `AccountInfo` from `remaining_accounts` and reads its balance using a raw accessor (`token::accessor::amount`) without first validating that the account is a legitimate SPL Token Account. While this appears to be a critical vulnerability allowing an attacker to fake their balance, the instruction is saved by a downstream check inside the `set_position_balance` function. This function verifies that the key of the provided account matches the token account address that was securely stored when the position was originally registered.

Although this downstream check currently prevents an exploit, it demonstrates the principle of defense in depth. Relying on complex downstream logic for security can be fragile. The instruction would be more robust and auditable if it performed top-level validation on the account from `remaining_accounts`, for example by attempting to deserialize it as an `Option<Account<'_, TokenAccount>>` and verifying its mint and owner immediately. There are many other cases where the top-level constraints are not enough, but checks in the `set_position_balance` function prevent vulnerabilities.

Adopting stricter top-level account constraints would make the programs safer, more readable, and easier to maintain over time by ensuring invariants are enforced at the account validation level rather than relying on downstream logic.

4.2. Recommendations for PDA seed design

Several PDA derivations in the codebase use similar seed structures without unique prefixes. While no exploitable cases were found, this pattern increases the risk of collision-based attacks in future code changes.

Without unique prefixes, PDA collisions could allow attackers to create accounts at addresses intended for legitimate operations, preventing users from creating necessary accounts and disrupting protocol functionality.

For example, in the margin-pool program, both account types use `[Pubkey, Pubkey]` patterns:

- Margin pool — `[airspace.key(), token_mint.key()]`
- Loan note account — `[margin_account.key(), loan_note_mint.key()]`

In the airspace program, both use the same prefix:

- `Airspace` — `[AIRSPACE, string_seed]`
- `AuthorityTransfer` — `[AIRSPACE, airspace_pubkey]`

This specific case is not very exploitable because it would require the airspace's 32-byte public key to be valid UTF-8 (extremely unlikely) and only the protocol governor can create airspaces.

We recommend using unique, constant prefixes as the first seed for each account type:

```
[b"margin-pool", airspace.key(), token_mint.key()]
[b"loan-note", margin_account.key(), loan_note_mint.key()]
[b"airspace", string_seed]
[b"authority-transfer", airspace_pubkey]
```

This approach prevents potential collisions and makes the codebase more maintainable.

4.3. Adapter-invoke trust model

The `adapter_invoke` instruction grants adapters full signing authority of the margin account through `invoke_signed()`. This creates a high-trust model where malicious or compromised adapters could drain any token accounts passed in the instruction. Several improvements could be made to help add defenses to prevent some malicious adapter behavior.

The system tracks `TokenBalanceChange` data from adapter invocations but completely ignores the return values in `adapter_invoke`, missing opportunities for validation. Consider adding slippage arguments that ensure that the user's token accounts do not experience unexpected or excessive changes in balance or value.

In addition, malicious adapters could approve themselves as delegates on token accounts without changing balances, then exploit this delegation at any time in the future to drain funds. Consider adding checks to verify that the delegation status of any token account passed in does not change.

4.4. Library error handling

The program-common library extensively uses `panic!()` and `.unwrap()` patterns in arithmetic, conversion, and other functions that are called with user-controlled data in the various programs. While these do not represent security vulnerabilities, they demonstrate poor error-handling patterns that could be improved.

For example, the `Number128` type implements standard arithmetic operators that panic on overflow:

```
impl Add for Number128 {
    fn add(self, rhs: Self) -> Self::Output {
        Number128::from_bits(self.bits.checked_add(rhs.bits).unwrap())
    }
}
```

In addition, the `Number` type also has conversion functions that explicitly panic:

```
pub fn as_u64(&self, exponent: i32) -> u64 {
    // ...
    if result < 0 || result > u64::MAX as i128 {
        panic!("Number overflow when converting to u64: {} with exponent {}",
    result, exponent);
    }
    result as u64
}
```

The library should consider rewriting the various panic-inducing functions into versions that return `Result` types.

This approach offers several significant advantages.

First, returning `Result` explicitly signals to the caller that an operation can fail, forcing them to handle potential errors gracefully. This promotes robust error handling, preventing unexpected program termination (panics) that can lead to denial of service or leave the system in an

inconsistent state.

Second, `Result` types provide clear, structured error information, allowing developers to understand why an operation failed and implement specific recovery logic. In contrast, panics often provide less actionable information, making debugging and error recovery more challenging.

Finally, using `Result` aligns with idiomatic Rust practices for recoverable errors, improving code clarity, maintainability, and composability.

4.5. Test suite

When building a complex protocol with cross-program interactions, liquidation mechanisms, and adapter integrations, comprehensive testing is essential. This includes testing for both positive and negative scenarios. Positive tests should verify that each instruction's side effects are as expected, while negative tests should cover every error condition, 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., compute unit costs)
- It displays code maturity
- It improves understanding of how the code functions, integrates, and operates — for developers and auditors alike

The margin program currently relies heavily on integration tests in `/tests/hosted/tests/` with minimal unit test coverage at the instruction level. While integration tests provide valuable end-to-end validation, they cannot comprehensively cover all edge cases and error conditions that unit tests would catch.

We recommend creating unit tests for instructions that currently lack direct testing, such as

- `register_position` — position registration logic and constraint validation
- `accounting_invoke` — permissionless adapter invocation without signer authority
- `admin_transfer_position` — emergency governor transfer functionality

In addition, the other programs like `airspace`, `lookup-table-registry`, and more would benefit from a comprehensive test suite.

5. 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 may not have 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.

Discriminant checks, ownership checks, and rent checks are not discussed for each individual account; unless otherwise stated, the program uses Anchor types, which perform the necessary checks automatically.

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.

5.1. Program: address-lookup-table-registry

Instruction: `init_registry_account`

This instruction initializes a new `RegistryAccount` PDA for the given `authority`.

Input parameters

None.

Accounts

- **`authority`**: The owner/authority of the registry account.
  - Signer: Yes.
  - Init: No.
  - PDA: No.
  - Mutable: No.
  - Constraints: None.
- **`payer`**: Pays for account creation and rent.
  - Signer: Yes.
  - Init: No.
  - PDA: No.
  - Mutable: Yes.
  - Constraints: Must have sufficient lamports to fund the registry account.
- **`registry_account`**: The registry PDA that will be created and initialized.

  - Signer: No.
  - Init: Yes.
  - PDA: Yes (seeds = `[authority.key.as_ref()]`, bump stored in `seed`).
  - Mutable: Yes.
  - Constraints: `space = 8 + size_of::<RegistryAccount>()`. Rent-exempt via `payer`. Created for `authority`.
- **`system_program`**: The system program.
  - Signer: No.
  - Init: No.
  - PDA: No.
  - Mutable: No.
  - Constraints: Must be the system program.

Additional checks and behavior

- Records the current slot in `last_created_slot`.
- Initializes internal fields: `version = 0`, `len = 0`, `capacity = 0`, and `tables = []`.
- Maximum registry capacity is bounded at compile time to 254 entries; no allocation beyond the base struct occurs during initialization.

CPI

None.

Instruction: `close_registry_account`

This instruction closes an empty `RegistryAccount` and transfers its lamports to `recipient`.

Input parameters

None.

Accounts

- **`registry_account`**: The registry account to close.
  - Signer: No.
  - Init: No.
  - PDA: Yes (same PDA derived at init).

    - Mutable: Yes.
    - Constraints: `has_one = authority`; `registry_account.len == 0` (otherwise, `RegistryNotEmpty`). Closed to `recipient`.
- **`authority`**: Authority of the registry account.
    - Signer: Yes.
    - Init: No.
    - PDA: No.
    - Mutable: No.
    - Constraints: Must match `registry_account.authority`.
- **`recipient`**: Receives lamports from the closed account.
    - Signer: No.
    - Init: No.
    - PDA: No.
    - Mutable: Yes.
    - Constraints: Unchecked; receives the closed funds.

Additional checks and behavior

- Enforces that no entries remain (`len == 0`) before closing.
- Uses Anchor's `close = recipient` to transfer remaining lamports.

CPI

None.

Instruction: `create_lookup_table`

This instruction creates a new address lookup table (ALT) via CPI to the Address Lookup Table program and records it in the registry. It grows the registry account as needed.

Input parameters

- `recent_slot: u64`: Slot used by the ALT program to derive the table address — also stored in `registry_account.last_created_slot`.
- `_discriminator: u64`: Ignored by this program. A fresh internal discriminator is computed instead.

Accounts

- **`authority`**: Authority of the registry account and ALT creator.
    - Signer: Yes.
    - Init: No.
    - PDA: No.
    - Mutable: No.
    - Constraints: Must match `registry_account.authority`.
- **`payer`**: Pays for ALT creation and any registry reallocation rent gap.
    - Signer: Yes.
    - Init: No.
    - PDA: No.
    - Mutable: Yes.
    - Constraints: Must have sufficient lamports to fund ALT creation (via CPI) and registry growth.
- **`registry_account`**: The owner's registry that tracks created ALTs.
    - Signer: No.
    - Init: No.
    - PDA: Yes.
    - Mutable: Yes.
    - Constraints: `registry_account.authority == authority.key()`.
- **`lookup_table`**: The ALT account being created.
    - Signer: No.
    - Init: No (created by the ALT program during CPI).
    - PDA: No (the ALT address is derived by the ALT program, not a PDA of this program).
    - Mutable: Yes.
    - Constraints: Will be validated by the ALT program; this program verifies the derived table address matches this account's key.
- **`address_lookup_table_program`**: The ALT program.
    - Signer: No.
    - Init: No.
    - PDA: No.
    - Mutable: No.
    - Constraints: Must equal `AddressLookupTab1e1111111111111111111111111`.
- **`system_program`**: The system program.
    - Signer: No.
    - Init: No.

- PDA: No.
- Mutable: No.
- Constraints: Must be the system program.

Additional checks and behavior

- Enforces a hard cap: if `registry_account.len as usize == MAX_REGISTRY_ENTRIES` (254), it returns `TooManyEntries`.
- Computes a nonzero entry discriminator as `discriminator::DEACTIVATED + 1` and rejects if `<= DEACTIVATED` with `InvalidDiscriminator`.
- Updates `registry_account.last_created_slot = recent_slot`.
- Handles dynamic growth. If `len == capacity`, it transfers the rent gap from `payer` and reallocates `registry_account` by `REGISTRY_ENTRY_SIZE`, then increments `len` by 1. Otherwise, it reuses an `EMPTY` slot found via `find_empty_entry()`. In both cases, it pushes or places a new `RegistryEntry { discriminator, table }` and increments `capacity` by 1.
- Validates that the ALT address derived by the CPI instruction equals the provided `lookup_table` account — otherwise, `InvalidLookupTable`.
- Errors with `InvalidState` if `len > capacity` — final safety check.

CPI

- Calls `address_lookup_table::instruction::create_lookup_table(authority, payer, recent_slot)` and `invoke` with `lookup_table`, `authority`, `payer`, `system_program`, and `address_lookup_table_program`.
- This CPI creates the ALT account at the derived address.

Instruction: `append_to_lookup_table`

This instruction appends a list of addresses to an existing ALT recorded in the registry.

Input parameters

- `addresses: Vec<Pubkey>`: Public keys to add to the ALT.

Accounts

- **`authority`**: Authority of the registry account and ALT authority.
  - Signer: Yes.
  - Init: No.

    - PDA: No.
    - Mutable: No.
    - Constraints: Must match `registry_account.authority`.
- **`payer`**: Pays for ALT extension if required by the ALT program.
    - Signer: Yes.
    - Init: No.
    - PDA: No.
    - Mutable: Yes.
    - Constraints: None beyond funding requirements.
- **`registry_account`**: Registry that must contain the ALT entry.
    - Signer: No.
    - Init: No.
    - PDA: Yes.
    - Mutable: No.
    - Constraints: `registry_account.authority == authority.key()`. Must contain an entry for `lookup_table` whose `discriminator > DEACTIVATED`.
- **`lookup_table`**: The ALT to be extended.
    - Signer: No.
    - Init: No.
    - PDA: No.
    - Mutable: Yes.
    - Constraints: Validated by the ALT program.
- **`address_lookup_table_program`**: The ALT program.
    - Signer: No.
    - Init: No.
    - PDA: No.
    - Mutable: No.
    - Constraints: Must equal `AddressLookupTab1e1111111111111111111111111`.
- **`system_program`**: The system program.
    - Signer: No.
    - Init: No.
    - PDA: No.
    - Mutable: No.
    - Constraints: Must be the system program.

Additional checks and behavior

- Looks up the ALT in `registry_account.tables` — if not found, `InvalidLookupTable`.

- Rejects appends to deactivated or empty entries (`discriminator <= DEACTIVATED`) with `InvalidDiscriminator`.

CPI

- Calls `address_lookup_table::instruction::extend_lookup_table(lookup_table, authority, Some(payer), addresses)` and `invoke` with `lookup_table`, `authority`, `payer`, `system_program`, and `address_lookup_table_program`.
- This CPI extends the ALT with the provided addresses.

Instruction: `remove_lookup_table`

This instruction removes an ALT from the registry by first deactivating it (if active) and, upon a subsequent call, closing it (if already deactivated). Also, it updates the registry's bookkeeping.

Input parameters

None.

Accounts

- **`authority`**: Authority of the registry and ALT.
  - Signer: Yes.
  - Init: No.
  - PDA: No.
  - Mutable: No.
  - Constraints: Must match `registry_account.authority`.
- **`recipient`**: Receives lamports when the ALT is closed.
  - Signer: Yes.
  - Init: No.
  - PDA: No.
  - Mutable: Yes.
  - Constraints: Will receive the closed ALT's lamports if/when closed.
- **`registry_account`**: The registry tracking the ALT entry.
  - Signer: No.
  - Init: No.
  - PDA: Yes.
  - Mutable: Yes.
  - Constraints: `registry_account.authority == authority.key()`.

- **`lookup_table`**: The ALT to deactivate or close.
    - Signer: No.
    - Init: No.
    - PDA: No.
    - Mutable: Yes.
    - Constraints: Validated by the ALT program.
- **`address_lookup_table_program`**: The ALT program.
    - Signer: No.
    - Init: No.
    - PDA: No.
    - Mutable: No.
    - Constraints: Must equal `AddressLookupTab1e1111111111111111111111111`.
- **`system_program`**: The system program.
    - Signer: No.
    - Init: No.
    - PDA: No.
    - Mutable: No.
    - Constraints: Must be the system program.

Additional checks and behavior

- Fetches a mutable registry entry for `lookup_table` — if not found, `InvalidLookupTable`.
- There are several state transitions:
    - If `discriminator == EMPTY`, error `InvalidState`.
    - If `discriminator > DEACTIVATED` (active), sets `discriminator = DEACTIVATED` and deactivates the ALT via CPI.
    - If `discriminator == DEACTIVATED`, marks entry as `EMPTY`, zeroes `table`, decrements `registry_account.len` (checked subtraction), and closes the ALT via CPI to `recipient`.
- Does not modify `capacity` — `len` reflects populated (non-`EMPTY`) entries.

CPI

- If deactivating, calls `address_lookup_table::instruction::deactivate_lookup_table(lookup_table, authority)` and `invoke` with `lookup_table`, `authority`, and `address_lookup_table_program`.
- If closing, calls `address_lookup_table::instruction::close_lookup_table(lookup_table,`

`authority, recipient)` and `invoke` with `lookup_table`, `authority`, `recipient`, `system_program`, and `address_lookup_table_program`.

5.2. Program: airspace

Instruction: `create_governor_id`

This instruction creates (or initializes if missing) the global `GovernorId` identity account that stores the protocol governor address.

Input parameters

None.

Accounts

- **`payer`**: Pays rent/fees to create or initialize the governor ID account.
  - Signer: Yes.
  - Init: No.
  - PDA: No.
  - Writable: Yes.
  - Constraints: None.
- **`governor_id`**: The global governor identity account.
  - Signer: No.
  - Init: Yes (`init_if_needed` — reinitialization is not performed; fields are only set if empty).
  - PDA: Yes (seed: `[GOVERNOR_ID]`).
  - Writable: Yes.
  - Constraints: `space = GovernorId::SIZE` and `payer = payer`.
- **`system_program`**: The system program.
  - Signer: No.
  - Init: No.
  - PDA: No.
  - Writable: No.
  - Constraints: Must be the system program.

Additional checks and behavior

- If `governor_id.governor` is unset (default pubkey), it is set to

    - `payer.key()` when the `testing` feature is enabled, and
    - `PROTOCOL_GOVERNOR_ID` otherwise.
- No changes are made if the account is already initialized with a nondefault governor.

CPI

- Implicit CPI to the system program for account creation when `init_if_needed` triggers.

Instruction: `governor_propose`

The current governor proposes a new governor. This instruction creates a one-off transfer record to be later finalized by the proposed governor.

Input parameters

- `proposed_governor: Pubkey`: The address to be set as the new governor upon finalization.

Accounts

- **`governor`**: The current governor.
    - Signer: Yes.
    - Init: No.
    - PDA: No.
    - Writable: Yes.
    - Constraints: Must match `governor_id.governor`.
- **`governor_id`**: The global governor identity account.
    - Signer: No.
    - Init: No.
    - PDA: Yes (seeded account, not created here).
    - Writable: Yes.
    - Constraints: `has_one = governor`.
- **`transfer`**: Temporary authority-transfer record.
    - Signer: No.
    - Init: Yes.
    - PDA: Yes (seeds: `[GOVERNOR_ID, governor_id.key()]`).
    - Writable: Yes.
    - Constraints: `space = 8 + size_of::<AuthorityTransfer>()` and `payer = governor`. Seed choice guarantees only one open proposal per `governor_id`.

- **`system_program`**: The system program.
    - Signer: No.
    - Init: No.
    - PDA: No.
    - Writable: No.
    - Constraints: Must be the system program.

Additional checks and behavior

- Writes `transfer.resource = governor_id.key()`, `transfer.current_authority = governor.key()`, and `transfer.new_authority = proposed_governor`.
- Emits `GovernorAuthorityTransferRequest`.

CPI

- Implicit CPI to the system program to create the `transfer` PDA.

Instruction: `governor_finalize_propose`

The proposed governor accepts the transfer and becomes the new governor. It closes the temporary transfer account, returning rent to the old governor.

Input parameters

None.

Accounts

- **`proposed_governor`**: The address that was proposed — must accept.
    - Signer: Yes.
    - Init: No.
    - PDA: No.
    - Writable: No.
    - Constraints: Must equal `transfer.new_authority`.
- **`old_governor`**: Receives rent from closing `transfer`.
    - Signer: No.
    - Init: No.
    - PDA: No.

- Writable: Yes.
- Constraints: Must equal `governor_id.governor` and `transfer.current_authority`.

- **`governor_id`**: The global governor identity account.
  - Signer: No.
  - Init: No.
  - PDA: Yes.
  - Writable: Yes.
  - Constraints: `governor_id.governor == old_governor.key()`.
- **`transfer`**: The pending transfer record.
  - Signer: No.
  - Init: No.
  - PDA: Yes (seeds: `[GOVERNOR_ID, governor_id.key()]`).
  - Writable: Yes.
  - Constraints: `transfer.resource == governor_id.key()`, `transfer.current_authority == old_governor.key()`, `transfer.new_authority == proposed_governor.key()`, and `close = old_governor`.

Additional checks and behavior

- Sets `governor_id.governor = transfer.new_authority`.
- Emits `GovernorAuthorityTransferCompleted`.
- Closes `transfer` to `old_governor`.

CPI

- None beyond Anchor's close semantics.

Instruction: `airspace_create`

This instruction creates a new `Airspace` account (an isolation domain). Only the governor must be valid; the authority of the airspace is set per input.

Input parameters

- `seed: String`: Seed string used in the airspace PDA derivation.
- `is_restricted: bool`: If true, permits are required and must be issued by authorized regulators.

- `authority: Pubkey`: The initial airspace authority.

Accounts

- **`payer`**: Pays rent for new airspace.
  - Signer: Yes.
  - Init: No.
  - PDA: No.
  - Writable: Yes.
  - Constraints: None.
- **`airspace`**: The airspace being created.
  - Signer: No.
  - Init: Yes.
  - PDA: Yes (seeds: `[AIRSPACE, seed.as_ref()]`).
  - Writable: Yes.
  - Constraints: `space = Airspace::SIZE` and `payer = payer`.
- **`governor`**: The current governor.
  - Signer: Yes.
  - Init: No.
  - PDA: No.
  - Writable: No.
  - Constraints: In nontesting builds, must match `governor_id.governor`.
- **`governor_id`**: Global governor identity.
  - Signer: No.
  - Init: No.
  - PDA: Yes.
  - Writable: No.
  - Constraints: `has_one = governor` (nontesting only).
- **`system_program`**: The system program.
  - Signer: No.
  - Init: No.
  - PDA: No.
  - Writable: No.
  - Constraints: Must be the system program.

Additional checks and behavior

- Sets `airspace.authority = authority` and `airspace.is_restricted =`

`is_restricted`.
- Emits `AirspaceCreated`, including the human-readable `seed`.

CPI

- Implicit CPI to the system program to create the airspace PDA.

Instruction: `airspace_propose_authority`

The current airspace authority proposes a new authority. This instruction creates a single pending transfer record scoped to the airspace.

Input parameters

- `proposed_authority: Pubkey`: Address to become the new authority upon finalization.

Accounts

- **`authority`**: Current airspace authority.
  - Signer: Yes.
  - Init: No.
  - PDA: No.
  - Writable: Yes.
  - Constraints: Must equal `airspace.authority`.
- **`airspace`**: Target airspace.
  - Signer: No.
  - Init: No.
  - PDA: Yes.
  - Writable: Yes.
  - Constraints: `has_one = authority`.
- **`transfer`**: Temporary authority-transfer record.
  - Signer: No.
  - Init: Yes.
  - PDA: Yes (seeds: `[AIRSPACE, airspace.key()]`).
  - Writable: Yes.
  - Constraints: `space = 8 + size_of::<AuthorityTransfer>()` and `payer = authority`. Seed choice ensures only one open transfer per airspace.
- **`system_program`**: The system program.

- Signer: No.
- Init: No.
- PDA: No.
- Writable: No.
- Constraints: Must be the system program.

Additional checks and behavior

- Writes `transfer.resource = airspace.key()`, `transfer.current_authority = authority.key()`, and `transfer.new_authority = proposed_authority`.
- Emits `AirspaceAuthorityTransfer`.

CPI

- Implicit CPI to the system program to create the `transfer` PDA.

Instruction: `airspace_finalize_authority`

The proposed authority accepts the transfer. The instruction updates the airspace authority and closes the transfer record.

Input parameters

None.

Accounts

- **`proposed_authority`**: The proposed new authority.
  - Signer: Yes.
  - Init: No.
  - PDA: No.
  - Writable: No.
  - Constraints: Must equal `transfer.new_authority`.
- **`old_authority`**: Receives rent when closing `transfer`.
  - Signer: No.
  - Init: No.
  - PDA: No.
  - Writable: Yes.

    - Constraints: Must equal `airspace.authority` and `transfer.current_authority`.
- **`airspace`**: Target airspace.
    - Signer: No.
    - Init: No.
    - PDA: Yes.
    - Writable: Yes.
    - Constraints: `airspace.authority == old_authority.key()`.
- **`transfer`**: Pending transfer record.
    - Signer: No.
    - Init: No.
    - PDA: Yes (seeds: `[AIRSPACE, airspace.key()]`).
    - Writable: Yes.
    - Constraints: `transfer.resource == airspace.key()`, `transfer.current_authority == old_authority.key()`, `transfer.new_authority == proposed_authority.key()`, and `close = old_authority`.

Additional checks and behavior

- Sets `airspace.authority = transfer.new_authority`.
- Emits `AirspaceAuthoritySet`.
- Closes `transfer` to `old_authority`.

CPI

- None beyond Anchor's close semantics.

Instruction: `airspace_permit_issuer_create`

Airspace authority grants a regulator license (`AirspacePermitIssuerId`) allowing the `issuer` to issue permits in this airspace.

Input parameters

- `issuer: Pubkey`: Address being authorized as a permit issuer (regulator).

Accounts

- **`payer`**: Pays rent to create the issuer ID account.
  - Signer: Yes.
  - Init: No.
  - PDA: No.
  - Writable: Yes.
  - Constraints: None.
- **`authority`**: Airspace authority.
  - Signer: Yes.
  - Init: No.
  - PDA: No.
  - Writable: No.
  - Constraints: Must equal `airspace.authority`.
- **`airspace`**: Target airspace.
  - Signer: No.
  - Init: No.
  - PDA: Yes.
  - Writable: No.
  - Constraints: `has_one = authority`.
- **`issuer_id`**: The regulator license account.
  - Signer: No.
  - Init: Yes.
  - PDA: Yes (seeds: `[AIRSPACE_PERMIT_ISSUER, airspace.key(), issuer]`).
  - Writable: Yes.
  - Constraints: `space = AirspacePermitIssuerId::SIZE` and `payer = payer`.
- **`system_program`**: The system program.
  - Signer: No.
  - Init: No.
  - PDA: No.
  - Writable: No.
  - Constraints: Must be the system program.

Additional checks and behavior

- Sets `issuer_id.airspace = airspace.key()` and `issuer_id.issuer = issuer`.
- Emits `AirspaceIssuerIdCreated`.

CPI

- Implicit CPI to the system program to create the `issuer_id` PDA.

Instruction: `airspace_permit_issuer_revoke`

Airspace authority revokes a regulator license; the license account is closed and rent is returned to `receiver`.

Input parameters

None.

Accounts

- **`receiver`**: Receives lamports from closing the issuer ID account.
  - Signer: Yes.
  - Init: No.
  - PDA: No.
  - Writable: Yes.
  - Constraints: None.
- **`authority`**: Airspace authority.
  - Signer: Yes.
  - Init: No.
  - PDA: No.
  - Writable: No.
  - Constraints: None (authority is validated via `airspace.has_one = authority`).
- **`airspace`**: Target airspace.
  - Signer: No.
  - Init: No.
  - PDA: Yes.
  - Writable: No.
  - Constraints: `has_one = authority`.
- **`issuer_id`**: Regulator license to be revoked.
  - Signer: No.
  - Init: No.
  - PDA: Yes (seeds: `[AIRSPACE_PERMIT_ISSUER, airspace.key(), issuer]`).

    - Writable: Yes.
    - Constraints: `has_one = airspace` and `close = receiver`.
- **`system_program`**: The system program.
    - Signer: No.
    - Init: No.
    - PDA: No.
    - Writable: No.
    - Constraints: Must be the system program.

Additional checks and behavior

- Emits `AirspaceIssuerIdRevoked { airspace, issuer }`.
- Closing sets the account's data length to 0 (so future lookups can detect revocation by zero data).

CPI

- None beyond Anchor's close semantics.

Instruction: `airspace_permit_create`

This instruction creates a user permit for an airspace. If the airspace is restricted and the caller is not the airspace authority, the caller must have a valid, nonrevoked issuer license.

Input parameters

- `owner: Pubkey`: The address receiving the permit.

Accounts

- **`payer`**: Pays rent for the new permit.
    - Signer: Yes.
    - Init: No.
    - PDA: No.
    - Writable: Yes.
    - Constraints: None.
- **`authority`**: Caller creating the permit (airspace authority or authorized issuer in restricted airspaces; anyone in unrestricted airspaces).

    - Signer: Yes.
    - Init: No.
    - PDA: No.
    - Writable: No.
    - Constraints: If `airspace.is_restricted` and `authority != airspace.authority`, must correspond to a valid `issuer_id` PDA below.
- **`airspace`**: Target airspace.
    - Signer: No.
    - Init: No.
    - PDA: Yes.
    - Writable: No.
    - Constraints: None.
- **`permit`**: The user permit account to create.
    - Signer: No.
    - Init: Yes.
    - PDA: Yes (seeds: `[AIRSPACE_PERMIT, airspace.key(), owner]`).
    - Writable: Yes.
    - Constraints: `space = AirspacePermit::SIZE` and `payer = payer`.
- **`issuer_id`**: Issuer license PDA corresponding to `authority`.
    - Signer: No.
    - Init: No.
    - PDA: Yes (seeds: `[AIRSPACE_PERMIT_ISSUER, airspace.key(), authority]`).
    - Writable: No.
    - Constraints: In restricted airspaces where `authority != airspace.authority`, this account must deserialize as `AirspacePermitIssuerId` (nonrevoked). In other cases, it is not used.
- **`system_program`**: The system program.
    - Signer: No.
    - Init: No.
    - PDA: No.
    - Writable: No.
    - Constraints: Must be the system program.

Additional checks and behavior

- If `airspace.is_restricted` and `authority != airspace.authority`, the instruction calls `try_deserialize` to deserialize `issuer_id` into `AirspacePermitIssuerId`. Failure indicates missing or revoked license and aborts.

- Sets the following:
  - `permit.airspace = airspace.key()`
  - `permit.owner = owner`
  - `permit.issuer = (airspace.authority == authority.key()) ? airspace.key() : authority.key()`
- Emits `AirspacePermitCreated`.

CPI

- Implicit CPI to the system program to create the `permit` PDA.

Instruction: `airspace_permit_revoke`

This instruction revokes a previously issued permit. It is always allowed by the airspace authority and by the original issuer. In restricted airspaces, if the issuer license has been revoked (the license account has zero data), *any* signer may revoke the permit.

Input parameters

None.

Accounts

- **`receiver`**: Receives lamports from closing the permit.
  - Signer: Yes.
  - Init: No.
  - PDA: No.
  - Writable: Yes.
  - Constraints: None.
- **`authority`**: Revoker — must be permitted per rules described above.
  - Signer: Yes.
  - Init: No.
  - PDA: No.
  - Writable: No.
  - Constraints: None — enforced directly and validated in handler logic.
- **`airspace`**: Airspace of the permit.
  - Signer: No.
  - Init: No.

    - PDA: Yes.
    - Writable: No.
    - Constraints: None.
- **`issuer_id`**: Issuer license PDA for `permit.issuer`.
    - Signer: No.
    - Init: No.
    - PDA: Yes (seeds: `[AIRSPACE_PERMIT_ISSUER, airspace.key(), permit.issuer]`).
    - Writable: No.
    - Constraints: May have zero data if previously revoked/closed — address still required to match seeds.
- **`permit`**: The permit to revoke (and close).
    - Signer: No.
    - Init: No.
    - PDA: Yes (seeds: `[AIRSPACE_PERMIT, airspace.key(), permit.owner]`).
    - Writable: Yes.
    - Constraints: `has_one = airspace` and `close = receiver`.

Additional checks and behavior

- Computes the following:
    - `is_airspace_authority = authority == airspace.authority`
    - `is_airspace_permit_issuer = authority == permit.issuer`
    - `is_restricted_issuer_not_revoked = airspace.is_restricted && !issuer_id.data_is_empty()`
- If *none* of the first two are true *and* `is_restricted_issuer_not_revoked` is true, it aborts with `PermissionDenied`. Effectively,
    - Airspace authority or the original issuer can always revoke.
    - In unrestricted airspaces, anyone can revoke.
    - In restricted airspaces where the issuer license has been revoked (the license data is empty), anyone can revoke.
- Emits `AirspacePermitRevoked`.
- Closes `permit` to `receiver`.

CPI

- None beyond Anchor's close semantics.

5.3. Program: margin-pool

Instruction: `create_pool`

This instruction creates a new `MarginPool` for a given `token_mint` within an `airspace` plus its vault token account, fee destination token account, and the Token-2022 note mints. Also, it creates and populates metadata for the underlying token and position tokens (deposit/loan notes).

Input parameters

None.

Accounts

- **`authority`**: Airspace authority authorized to create pools.
  - Signer: Yes.
  - Init: No.
  - PDA: No.
  - Mutable: No.
  - Constraints: Must equal `airspace.authority`.
- **`airspace`**: Airspace in which the pool is registered.
  - Signer: No.
  - Init: No.
  - PDA: No.
  - Mutable: No.
  - Constraints: `airspace.authority == authority`.
- **`fee_owner`**: The owner that will control fee withdrawals.
  - Signer: Yes.
  - Init: No.
  - PDA: No.
  - Mutable: No.
  - Constraints: Proves existence and authority over fee destination.
- **`payer`**: Pays rent for newly created accounts.
  - Signer: Yes.
  - Init: No.
  - PDA: No.
  - Mutable: Yes.
  - Constraints: Must have sufficient lamports.
- **`margin_pool`**: The pool account to create.

  - Signer: No.
  - Init: Yes (created via `system_program::create_account`).
  - PDA: Yes (`[airspace, token_mint]`).
  - Mutable: Yes.
  - Constraints: Space = `8 + size_of::<MarginPool>()` and owner = program ID.
- **`vault`**: Pool vault token account for `token_mint`.
  - Signer: No.
  - Init: Yes (created in handler).
  - PDA: Yes (`[margin_pool, "vault"]`).
  - Mutable: Yes.
  - Constraints: Token account for `token_mint` — authority = `margin_pool`.
- **`deposit_note_mint`**: Token-2022 mint for deposit notes.
  - Signer: No.
  - Init: Yes (via Anchor `#[account(init)]`).
  - PDA: Yes (`[margin_pool, "deposit-notes"]`).
  - Mutable: Yes.
  - Constraints: Decimals = `token_mint.decimals`, mint authority = `margin_pool`, and token program = `pool_token_program`.
- **`loan_note_mint`**: Token-2022 mint for loan notes.
  - Signer: No.
  - Init: Yes (via Anchor `#[account(init)]`).
  - PDA: Yes (`[margin_pool, "loan-notes"]`).
  - Mutable: Yes.
  - Constraints: Decimals = `token_mint.decimals`, mint authority = `margin_pool`, and token program = `pool_token_program`.
- **`token_mint`**: Underlying SPL/Token-2022 mint that the pool uses.
  - Signer: No.
  - Init: No.
  - PDA: No.
  - Mutable: No.
  - Constraints: Must pass mint-extension validation (no disabled/fee extensions).
- **`fee_destination`**: Token-2022 token account holding collected fees (in deposit notes).
  - Signer: No.
  - Init: Yes (created in handler).
  - PDA: Yes (`[margin_pool, fee_owner, "margin-pool-fee-destination"]`).
  - Mutable: Yes.
  - Constraints: Token account for `deposit_note_mint`, authority = `fee_owner`,

and token program = `pool_token_program`.

- **`token_metadata`**: Metadata account for the underlying token.
  - Signer: No.
  - Init: Yes (via CPI to metadata program).
  - PDA: No.
  - Mutable: Yes.
  - Constraints: Created/updated via metadata program CPI.
- **`deposit_note_metadata`**: Metadata account for deposit notes.
  - Signer: No.
  - Init: Yes (via CPI).
  - PDA: No.
  - Mutable: Yes.
  - Constraints: Created/updated via metadata program CPI.
- **`loan_note_metadata`**: Metadata account for loan notes.
  - Signer: No.
  - Init: Yes (via CPI).
  - PDA: No.
  - Mutable: Yes.
  - Constraints: Created/updated via metadata program CPI.
- **`mint_token_program`**: Token program for the underlying mint (`spl-token` or Token-2022).
  - Signer: No.
  - Init: No.
  - PDA: No.
  - Mutable: No.
- **`pool_token_program`**: Token-2022 program used by note mints and note token accounts.
  - Signer: No.
  - Init: No.
  - PDA: No.
  - Mutable: No.
- **`metadata_program`**: Metadata program for on-chain metadata entries.
  - Signer: No.
  - Init: No.
  - PDA: No.
  - Mutable: No.
- **`system_program`**: The system program for account creation.
  - Signer: No.

- Init: No.
- PDA: No.
- Mutable: No.

Additional checks and behavior

- Validates `token_mint` extensions, disallows certain Token-2022 extensions and nonzero transfer fees, and errors with `TokenExtensionNotEnabled` if invalid.
- Creates `vault` and `fee_destination` token accounts and initializes them with `InitializeAccount3`.
- Creates and initializes `margin_pool`; sets version, addresses, mints, fee destination, and timestamps; and writes discriminator and serialized data.
- Initializes and sets metadata for deposit/loan position tokens and the underlying token. Emits `PoolCreated` and metadata configured events.

CPI

- `system_program::create_account` for pool account and token accounts.
- `token_interface::initialize_account3` for `vault` and `fee_destination`.
- `glow_metadata::cpi::{create_entry,set_entry}` to create and populate metadata accounts.

Instruction: `collect`

This instruction accrues interest up to a bounded window and mints deposit-note fees to the pool's `fee_destination`.

Input parameters

None.

Accounts

- **`margin_pool`**: Pool to collect fees from.
  - Signer: No.
  - Init: No.
  - PDA: No.
  - Mutable: Yes.
  - Constraints: `has_one = vault`, `has_one = deposit_note_mint`, and `has_one = fee_destination`.

- **`vault`**: Pool's underlying token vault.
    - Signer: No.
    - Init: No.
    - PDA: No.
    - Mutable: Yes (read for balances, SPL state mutated when minting elsewhere, and kept mutable by constraint).
    - Constraints: Token account for underlying mint — authority = `margin_pool`.
- **`fee_destination`**: Deposit-note token account receiving minted fee notes.
    - Signer: No.
    - Init: No.
    - PDA: No.
    - Mutable: Yes.
- **`deposit_note_mint`**: Deposit-note mint.
    - Signer: No.
    - Init: No.
    - PDA: No.
    - Mutable: Yes.
- **`token_program`**: Token interface (Token-2022 for note mint).
    - Signer: No.
    - Init: No.
    - PDA: No.
    - Mutable: No.

Additional checks and behavior

- Accrues interest up to `MAX_ACCRUAL_SECONDS` — if the clock drift exceeds the cap and accrual could not fully catch up, logs and exits early.
- Calls `collect_accrued_fees()` to compute notes to mint and updates pool-fee accounting.
- Mints fee notes to `fee_destination` and emits `Collect`, including postcollection balances.

CPI

- `token_interface::mint_to` to mint deposit notes to `fee_destination`.

Instruction: `configure`

This instruction updates pool configuration, oracle settings, and/or token metadata for deposit/loan notes.

Input parameters

- `metadata: Option<TokenMetadataParams>`: Struct of token metadata fields to apply to deposit and loan position tokens.
    - `token_kind: TokenKind`
    - `collateral_weight: u16`
    - `max_leverage: u16`
    - `max_staleness: u64`
    - `token_features: u16` (bitflags validated for compatibility)
- `config: Option<MarginPoolConfig>`: New pool config — monotonicity of utilization/borrow rate steps and fee caps are enforced.
- `oracle: Option<TokenPriceOracle>`: New oracle configuration (kept in sync with token metadata).

Accounts

- **`payer`**: Pays any CPI metadata writes.
    - Signer: Yes.
    - Init: No.
    - PDA: No.
    - Mutable: Yes.
- **`authority`**: Pool/airspace authority.
    - Signer: Yes.
    - Init: No.
    - PDA: No.
    - Mutable: No.
- **`airspace`**: Airspace the pool belongs to.
    - Signer: No.
    - Init: No.
    - PDA: No.
    - Mutable: No.
    - Constraints: `airspace.authority == authority`.
- **`margin_pool`**: Pool to configure.
    - Signer: No.

  - Init: No.
  - PDA: No.
  - Mutable: Yes.
  - Constraints: `has_one = token_mint` and `margin_pool.airspace == airspace`.
- **`token_mint`**: Underlying token mint.
  - Signer: No.
  - Init: No.
  - PDA: No.
  - Mutable: No.
- **`token_metadata`**: Existing metadata for the underlying token.
  - Signer: No.
  - Init: No.
  - PDA: No.
  - Mutable: Yes.
  - Constraints: `has_one = token_mint` and `has_one = airspace`.
- **`deposit_metadata`**: Metadata for deposit notes.
  - Signer: No.
  - Init: No.
  - PDA: No.
  - Mutable: Yes.
  - Constraints: `has_one = airspace`, `underlying_token_mint == token_mint`, and `position_token_mint == margin_pool.deposit_note_mint == deposit_note_mint`.
- **`loan_metadata`**: Metadata for loan notes.
  - Signer: No.
  - Init: No.
  - PDA: No.
  - Mutable: Yes.
  - Constraints: `has_one = airspace`, `underlying_token_mint == token_mint`, and `position_token_mint == margin_pool.loan_note_mint == loan_note_mint`.
- **`deposit_note_mint`**: Key used for deposit-metadata entry indexing.
  - Signer: No.
  - Init: No.
  - PDA: No.
  - Mutable: No.
- **`loan_note_mint`**: Key used for loan-metadata entry indexing.

  - Signer: No.
  - Init: No.
  - PDA: No.
  - Mutable: No.
- **`metadata_program`** and **`system_program`**: Programs for metadata and system ops.
  - Signer: No.
  - Init: No.
  - PDA: No.
  - Mutable: No.

Additional checks and behavior

- If `config` is present, enforces
  - `utilization_rate_1 < utilization_rate_2`,
  - `borrow_rate_0 < borrow_rate_1 < borrow_rate_2 < borrow_rate_3`, and
  - `management_fee_rate <= MAX_MANAGEMENT_FEE_RATE`.
- If `oracle` is present, sets `pool.token_price_oracle`, updates `token_metadata`, and emits `TokenMetadataConfigured`.
- Emits `PoolConfigured` with provided or default values.
- If `metadata` is present, validates `max_staleness <= MAX_TOKEN_STALENESS` and `token_features` compatibility, writes both deposit and loan position metadata, and emits `PositionTokenMetadataConfigured` events.

CPI

- `glow_metadata::cpi::set_entry` for token and position metadata updates.

Instruction: `deposit`

This instruction deposits underlying tokens into the pool and mints deposit notes to the destination.

Input parameters

- `change_kind: ChangeKind`: How to interpret `amount` (set/shift semantics).
- `amount: u64`: Tokens to apply under `change_kind`.

Accounts

- **`margin_pool`**: Target pool.

    - Signer: No.
    - Init: No.
    - PDA: No.
    - Mutable: Yes.
    - Constraints: `has_one = vault` and `has_one = deposit_note_mint`.
- **`vault`**: Pool vault for underlying tokens.
    - Signer: No.
    - Init: No.
    - PDA: Yes (`[margin_pool, "vault"]`).
    - Mutable: Yes.
- **`deposit_note_mint`**: Deposit-note mint (Token-2022).
    - Signer: No.
    - Init: No.
    - PDA: Yes (`[margin_pool, "deposit-notes"]`).
    - Mutable: Yes.
- **`depositor`**: Authority moving tokens.
    - Signer: Yes.
    - Init: No.
    - PDA: No.
    - Mutable: No.
- **`source`**: User token account of `source_mint`.
    - Signer: No.
    - Init: No.
    - PDA: No.
    - Mutable: Yes.
    - Constraints: Mint = `source_mint`, authority = `depositor`, and token program = `mint_token_program`.
- **`source_mint`**: Underlying token mint.
    - Signer: No.
    - Init: No.
    - PDA: No.
    - Mutable: No.
- **`destination`**: Token account to receive deposit notes.
    - Signer: No.
    - Init: No.
    - PDA: No.
    - Mutable: Yes.
    - Constraints: Mint = `deposit_note_mint` and token program =

`pool_token_program`.

- **`mint_token_program`** and **`pool_token_program`**: Token interfaces for underlying and notes.
  - Signer: No.
  - Init: No.
  - PDA: No.
  - Mutable: No.
- **`deposit_metadata`**: Position token configuration for restrictions.
  - Signer: No.
  - Init: No.
  - PDA: No.
  - Mutable: No.
  - Constraints: `position_token_mint == deposit_note_mint`.
- **`optional_margin_account`**: Required if token features include `RESTRICTED`.
  - Signer: No.
  - Init: No.
  - PDA: No.
  - Mutable: No.

Additional checks and behavior

- Accrues interest and fails with `InterestAccrualBehind` if too far behind.
- If `RESTRICTED`,
  - requires `optional_margin_account` present, and
  - validates `margin_account.owner == depositor`, `margin_account.airspace == pool.airspace`, `destination.owner == margin_account`, and token-feature compatibility with the margin account.
- Computes deposit amounts using exchange rate and `change_kind`, enforces deposit limit, updates pool accounting, transfers underlying to `vault`, and mints notes to `destination`.
- Emits `Deposit`.

CPI

- `token_interface::transfer_checked` from `source` to `vault`.
- `token_interface::mint_to` to `destination`.

Instruction: `withdraw`

This instruction burns deposit notes and transfers underlying tokens from the pool vault to the destination.

Input parameters

- `change_kind: ChangeKind`
- `amount: u64`

Accounts

- **`depositor`**: Authority burning deposit notes or margin-account signer.
  - Signer: Yes.
  - Init: No.
  - PDA: No.
  - Mutable: No.
- **`margin_pool`**: Pool to withdraw from.
  - Signer: No.
  - Init: No.
  - PDA: No.
  - Mutable: Yes.
  - Constraints: `has_one = vault` and `has_one = deposit_note_mint`.
- **`vault`**: Pool vault token account.
  - Signer: No.
  - Init: No.
  - PDA: Yes.
  - Mutable: Yes.
- **`deposit_note_mint`**: Deposit-note mint.
  - Signer: No.
  - Init: No.
  - PDA: Yes.
  - Mutable: Yes.
- **`source`**: User's deposit-note token account.
  - Signer: No.
  - Init: No.
  - PDA: No.
  - Mutable: Yes.

    - Constraints: Must be owned by the burning authority — validated by the token program.
- **`destination`**: Token account to receive underlying tokens.
    - Signer: No.
    - Init: No.
    - PDA: No.
    - Mutable: Yes.
    - Constraints: Mint = `token_mint` and token program = `mint_token_program`.
- **`token_mint`**: Underlying mint.
    - Signer: No.
    - Init: No.
    - PDA: No.
    - Mutable: No.
- **`mint_token_program`** and **`pool_token_program`**: Token interfaces (underlying and notes).
    - Signer: No.
    - Init: No.
    - PDA: No.
    - Mutable: No.

Additional checks and behavior

- Validates the allowed `destination.owner`. If `depositor` is a `MarginAccount`, `destination.owner` must be that margin account or its `owner`. Otherwise, `destination.owner` must be `depositor`.
- Accrues interest and fails if too far behind.
- Converts/bounds withdraw amounts, updates pool, transfers underlying from `vault`, and burns deposit notes from `source`.
- Emits `Withdraw`.

CPI

- `token_interface::transfer_checked` from `vault` to `destination` (signed by pool PDA).
- `token_interface::burn` on `source` (authority = `depositor`).

Instruction: `margin_borrow`

This instruction borrows underlying via the pool and immediately deposits it back to receive deposit notes to the margin account's note account. It effectively internalizes the borrowed tokens as deposit notes.

Input parameters

- `change_kind: ChangeKind`
- `amount: u64`

Accounts

- **`margin_account`**: Margin account being executed on.
  - Signer: Yes.
  - Init: No.
  - PDA: No.
  - Mutable: No.
- **`margin_pool`**: Pool to borrow from and deposit into.
  - Signer: No.
  - Init: No.
  - PDA: No.
  - Mutable: Yes.
  - Constraints: `has_one = loan_note_mint` and `has_one = deposit_note_mint`.
- **`loan_note_mint`**: Loan-note mint (Token-2022).
  - Signer: No.
  - Init: No.
  - PDA: Yes.
  - Mutable: Yes.
  - Constraints: Token program = `pool_token_program`.
- **`deposit_note_mint`**: Deposit-note mint (Token-2022).
  - Signer: No.
  - Init: No.
  - PDA: Yes.
  - Mutable: Yes.
  - Constraints: Token program = `pool_token_program`.
- **`loan_account`**: Loan-note token account, owned by the pool.
  - Signer: No.

- Init: No.
- PDA: Yes (`[margin_account, loan_note_mint]`).
- Mutable: Yes.
- Constraints: `owner == margin_pool`.

- **`deposit_account`**: Deposit-note token account owned by the margin account.
  - Signer: No.
  - Init: No.
  - PDA: No.
  - Mutable: Yes.
  - Constraints: `owner == margin_account`.
- **`pool_token_program`**: Token-2022 interface.
  - Signer: No.
  - Init: No.
  - PDA: No.
  - Mutable: No.

Additional checks and behavior

- Accrues interest and computes `borrow_amount` from `change_kind`.
- Enforces lending enabled, utilization, and borrow limits and updates pool borrowed/deposit accounting.
- Mints loan notes to `loan_account` and mints deposit notes to `deposit_account`.
- Notifies margin program of a token balance change (`Borrow`) and emits `MarginBorrow`.

CPI

- `token_interface::mint_to` to `loan_account` and `deposit_account`.

Instruction: `margin_borrow_v2`

This instruction borrows underlying via the pool and transfers the underlying tokens directly to a destination token account (no immediate deposit).

Input parameters

- `amount: u64`

Accounts

- **`margin_account`**: Margin account being executed on.
    - Signer: Yes.
    - Init: No.
    - PDA: No.
    - Mutable: No.
- **`margin_pool`**: Pool to borrow from.
    - Signer: No.
    - Init: No.
    - PDA: No.
    - Mutable: Yes.
    - Constraints: `has_one = vault` and `has_one = loan_note_mint`.
- **`vault`**: Pool vault for underlying tokens.
    - Signer: No.
    - Init: No.
    - PDA: Yes.
    - Mutable: Yes.
- **`loan_note_mint`**: Loan-note mint.
    - Signer: No.
    - Init: No.
    - PDA: Yes.
    - Mutable: Yes.
- **`token_mint`**: Underlying mint (for decimals).
    - Signer: No.
    - Init: No.
    - PDA: No.
    - Mutable: No.
- **`loan_account`**: Loan-note account owned by the pool for this margin account.
    - Signer: No.
    - Init: No.
    - PDA: Yes (`[margin_account, loan_note_mint]`).
    - Mutable: Yes.
    - Constraints: `owner == margin_pool`.
- **`destination`**: Token account receiving the borrowed underlying.
    - Signer: No.
    - Init: No.

    - PDA: No.
    - Mutable: Yes.
- **`mint_token_program`**: Token interface for underlying.
    - Signer: No.
    - Init: No.
    - PDA: No.
    - Mutable: No.
- **`pool_token_program`**: Token-2022 program for note minting.
    - Signer: No.
    - Init: No.
    - PDA: No.
    - Mutable: No.

Additional checks and behavior

- Accrues interest and computes loan `borrow_amount`.
- Updates pool accounting, mints loan notes to `loan_account`, and transfers underlying from `vault` to `destination`.
- Notifies margin program of `Borrow` and emits `MarginBorrow` (with `deposit_notes = 0`).

CPI

- `token_interface::mint_to` to `loan_account`.
- `token_interface::transfer_checked` from `vault` to `destination` (signed by pool PDA).

Instruction: `margin_repay`

This instruction repays a margin loan using deposit notes held by the margin account. It burns loan notes and burns deposit notes corresponding to the withdrawn underlying used for repayment.

Input parameters

- `change_kind: ChangeKind`
- `amount: u64`

Accounts

- **`margin_account`**: Margin account being executed on.

    - Signer: Yes.
    - Init: No.
    - PDA: No.
    - Mutable: No.
- **`margin_pool`**: Pool with the outstanding loan.
    - Signer: No.
    - Init: No.
    - PDA: No.
    - Mutable: Yes.
    - Constraints: `has_one = deposit_note_mint` and `has_one = loan_note_mint`.
- **`loan_note_mint`**: Loan-note mint.
    - Signer: No.
    - Init: No.
    - PDA: Yes.
    - Mutable: Yes.
- **`deposit_note_mint`**: Deposit-note mint.
    - Signer: No.
    - Init: No.
    - PDA: Yes.
    - Mutable: Yes.
- **`loan_account`**: Loan-note token account.
    - Signer: No.
    - Init: No.
    - PDA: No.
    - Mutable: Yes.
- **`deposit_account`**: Deposit-note token account owned by the margin account.
    - Signer: No.
    - Init: No.
    - PDA: No.
    - Mutable: Yes.
- **`pool_token_program`**: Token interface for note burns (Token-2022).
    - Signer: No.
    - Init: No.
    - PDA: No.
    - Mutable: No.

Additional checks and behavior

- Accrues interest, computes `repay_amount` from deposit notes and outstanding loan, and computes `withdraw_amount` of deposit notes needed to cover token repayment.
- Updates pool via `margin_repay()` (burns loan/deposit notes accounting and reduces borrowed principal).
- Burns loan notes (authority = pool PDA) and burns deposit notes (authority = `margin_account`).
- Notifies margin program of `Repay` and emits `MarginRepay`.

CPI

- `token_interface::burn` on `loan_account` (signed by pool PDA).
- `token_interface::burn` on `deposit_account` (authority = `margin_account`).

Instruction: `repay`

This instruction repays a margin loan using external underlying tokens. It transfers underlying into the vault and burns the corresponding loan notes.

Input parameters

- `change_kind: ChangeKind`
- `amount: u64`

Accounts

- **`margin_pool`**: Pool with the outstanding loan.
  - Signer: No.
  - Init: No.
  - PDA: No.
  - Mutable: Yes.
  - Constraints: `has_one = loan_note_mint` and `has_one = vault`.
- **`loan_note_mint`**: Loan-note mint.
  - Signer: No.
  - Init: No.
  - PDA: Yes.
  - Mutable: Yes.
- **`token_mint`**: Underlying mint (for decimals).

    - Signer: No.
    - Init: No.
    - PDA: No.
    - Mutable: No.
- **`vault`**: Pool vault for underlying.
    - Signer: No.
    - Init: No.
    - PDA: Yes.
    - Mutable: Yes.
- **`loan_account`**: Borrower's loan-note token account.
    - Signer: No.
    - Init: No.
    - PDA: No.
    - Mutable: Yes.
- **`repayment_token_account`**: Token account holding underlying to repay.
    - Signer: No.
    - Init: No.
    - PDA: No.
    - Mutable: Yes.
- **`repayment_account_authority`**: Authority over `repayment_token_account`.
    - Signer: Yes.
    - Init: No.
    - PDA: No.
    - Mutable: No.
- **`mint_token_program`** and **`pool_token_program`**: Token interfaces for underlying and notes.
    - Signer: No.
    - Init: No.
    - PDA: No.
    - Mutable: No.

Additional checks and behavior

- Accrues interest and computes the repay amounts from available tokens and outstanding notes.
- Updates pool accounting, transfers tokens into `vault`, and burns loan notes (signed by pool PDA).
- Returns serialized `AdapterResult` via Solana return data indicating a `Repay` token

balance change and emits `Repay`.

CPI

- `token_interface::transfer_checked` from `repayment_token_account` to `vault`.
- `token_interface::burn` on `loan_account` (signed by pool PDA).

Instruction: `withdraw_fees`

This instruction converts fee notes held in the pool's `fee_destination` into underlying tokens and withdraws them to a fee owner's token account, burning the fee notes.

Input parameters

None.

Accounts

- **`fee_owner`**: Fee owner withdrawing fees.
  - Signer: Yes.
  - Init: No.
  - PDA: No.
  - Mutable: No.
- **`margin_pool`**: Pool to withdraw fees from.
  - Signer: No.
  - Init: No.
  - PDA: No.
  - Mutable: No.
  - Constraints: `has_one = vault`, `has_one = deposit_note_mint`, and `has_one = fee_destination`.
- **`vault`**: Pool vault holding underlying tokens.
  - Signer: No.
  - Init: No.
  - PDA: Yes.
  - Mutable: Yes.
- **`fee_destination`**: Fee-note token account owned by `fee_owner`.
  - Signer: No.
  - Init: No.

    - PDA: Yes.
    - Mutable: Yes.
    - Constraints: Mint = `deposit_note_mint`, authority = `fee_owner`, and token program = `pool_token_program`.
- **`fee_withdrawal_destination`**: Recipient token account for withdrawn underlying.
    - Signer: No.
    - Init: No.
    - PDA: No.
    - Mutable: Yes.
    - Constraints: Mint = `token_mint`, authority = `fee_owner`, and token program = `mint_token_program`.
- **`token_mint`**: Underlying mint.
    - Signer: No.
    - Init: No.
    - PDA: No.
    - Mutable: No.
- **`deposit_note_mint`**: Deposit-note mint.
    - Signer: No.
    - Init: No.
    - PDA: Yes.
    - Mutable: Yes.
- **`mint_token_program`** and **`pool_token_program`**: Token interfaces for underlying and notes.
    - Signer: No.
    - Init: No.
    - PDA: No.
    - Mutable: No.

Additional checks and behavior

- Converts all fee notes in `fee_destination` to underlying at the current exchange rate.
- Transfers underlying from `vault` to `fee_withdrawal_destination` (signed by pool PDA) and burns the fee notes from `fee_destination` (authority = `fee_owner`).
- Emits `FeesWithdrawn`.

CPI

- `token_interface::transfer_checked` from `vault` to `fee_withdrawal_destination`.
- `token_interface::burn` on `fee_destination`.

Instruction: `register_loan`

This instruction creates the PDA loan-note token account for a margin account and requests the margin program to register the position.

Input parameters

None.

Accounts

- **`margin_account`**: Margin account authority.
  - Signer: Yes.
  - Init: No.
  - PDA: No.
  - Mutable: No.
- **`loan_token_config`**: Auxiliary config account required by margin (opaque here).
  - Signer: No.
  - Init: No.
  - PDA: No.
  - Mutable: No.
- **`loan_note_account`**: PDA token account to store loan notes.
  - Signer: No.
  - Init: Yes (via Anchor).
  - PDA: Yes (`[margin_account, loan_note_mint]`).
  - Mutable: Yes.
  - Constraints: Mint = `loan_note_mint`, authority = `margin_pool`, and token program = `token_program`.
- **`loan_note_mint`**: Loan-note mint.
  - Signer: No.
  - Init: No.
  - PDA: Yes.
  - Mutable: No.
- **`margin_pool`**: Pool that issues the loan notes.
  - Signer: No.
  - Init: No.
  - PDA: No.
  - Mutable: No.

    - Constraints: `has_one = loan_note_mint`.
- **`payer`**: Rent payer for the created token account.
    - Signer: Yes.
    - Init: No.
    - PDA: No.
    - Mutable: Yes.
- **`token_program`**: Token-2022 program.
    - Signer: No.
    - Init: No.
    - PDA: No.
    - Mutable: No.
- **`system_program`** and **`rent`**: The system and rent sysvar.
    - Signer: No.
    - Init: No.
    - PDA: No.
    - Mutable: No.

Additional checks and behavior

- After creating the loan-note token account, writes an adapter result requesting `PositionChange::Register` so the margin program can register the position.

CPI

- Anchor initializes the token account via Token-2022.
- `glow_margin::write_adapter_result` to notify margin program (via account data/return data semantics).

Instruction: `close_loan`

This instruction loses a previously created loan-note token account for a margin position and informs the margin program that the position is closed.

Input parameters

None.

Accounts

- **`margin_account`**: Margin account being executed on.
  - Signer: Yes.
  - Init: No.
  - PDA: No.
  - Mutable: No.
- **`loan_note_account`**: PDA token account holding loan notes to be closed.
  - Signer: No.
  - Init: No.
  - PDA: Yes (`[margin_account, loan_note_mint]`).
  - Mutable: Yes.
- **`loan_note_mint`**: Loan-note mint.
  - Signer: No.
  - Init: No.
  - PDA: Yes.
  - Mutable: No.
- **`margin_pool`**: Pool that owns the loan-note account authority.
  - Signer: No.
  - Init: No.
  - PDA: No.
  - Mutable: No.
  - Constraints: `has_one = loan_note_mint`.
- **`beneficiary`**: Recipient of reclaimed rent from closing the token account.
  - Signer: Yes.
  - Init: No.
  - PDA: No.
  - Mutable: Yes.
- **`token_program`**: Token-2022 program.
  - Signer: No.
  - Init: No.
  - PDA: No.
  - Mutable: No.

Additional checks and behavior

- Closes the loan-note token account with authority `margin_pool` (signed by pool PDA) and sends lamports to `beneficiary`.

- Notifies margin program of `PositionChange::Close`.

CPI

- `token_interface::close_account` on `loan_note_account`.
- `glow_margin::write_adapter_result` to notify margin program.

Instruction: `admin_transfer_loan`

This instruction has an administrative function; it moves loan notes between two pool-controlled loan-note token accounts.

Input parameters

- `amount: u64`

Accounts

- **`authority`**: Protocol governor.
  - Signer: Yes.
  - Init: No.
  - PDA: No.
  - Mutable: No.
  - Constraints: Must equal `PROTOCOL_GOVERNOR_ID`.
- **`margin_pool`**: Pool owning the loan-note accounts.
  - Signer: No.
  - Init: No.
  - PDA: No.
  - Mutable: No.
- **`source_loan_account`**: Source loan-note token account.
  - Signer: No.
  - Init: No.
  - PDA: No.
  - Mutable: Yes.
  - Constraints: Authority = `margin_pool`.
- **`target_loan_account`**: Target loan-note token account.
  - Signer: No.
  - Init: No.

    - PDA: No.
    - Mutable: Yes.
    - Constraints: Authority = `margin_pool`.
- **`loan_note_mint`**: Loan-note mint (for decimals).
    - Signer: No.
    - Init: No.
    - PDA: Yes.
    - Mutable: No.
- **`pool_token_program`**: Token-2022 interface.
    - Signer: No.
    - Init: No.
    - PDA: No.
    - Mutable: No.

Additional checks and behavior

- Transfers `amount` of loan notes between accounts using `margin_pool` signer PDA and emits `LoanTransferred`.

CPI

- `token_interface::transfer_checked` from source to target (signed by pool PDA).

Instruction: `margin_refresh_position`

This instruction fetches price data from Pyth Pull Receiver accounts and reports deposit/loan note prices back to the margin program for the position.

Input parameters

None.

Accounts

- **`margin_account`**: Margin account whose position is being refreshed.
    - Signer: No.
    - Init: No.
    - PDA: No.

    - Mutable: No.
- **`margin_pool`**: Pool whose pricing is being refreshed.
    - Signer: No.
    - Init: No.
    - PDA: No.
    - Mutable: No.
- **`price_oracle`**: Pyth Pull Receiver price update account for the pool token or redemption rate feed.
    - Signer: No.
    - Init: No.
    - PDA: No.
    - Mutable: No.
    - Constraints: Owner must be the Pyth Pull Receiver program (or test program on devnet).
- **`redemption_quote_oracle`**: Optional Pyth Pull Receiver price update for the quote token when using redemption rates.
    - Signer: No.
    - Init: No.
    - PDA: No.
    - Mutable: No.
    - Constraints: Required when pool oracle is `PythPullRedemption` — owner must be Pyth Pull Receiver (or test program).

Additional checks and behavior

- Verifies oracle account owners to prevent spoofed oracle accounts.
- Deserializes `PriceUpdateV2` from provided accounts.
- If the pool uses `PythPullRedemption`, also consumes the quote oracle and composes the price and confidence in quote terms.
- Computes deposit/loan note prices, confidence, and TWAP using current exchange rates — and publishes via adapter result to the margin program for both note mints.

CPI

- `glow_margin::write_adapter_result` to pass `PositionChange::Price` info (two entries: deposit and loan notes).

5.4. Program: margin

Instruction: `create_account`

This instruction creates a new margin account for a user within a specific airspace. The account is initialized with owner permissions and feature flags that control certain behaviors.

Input parameters

- The `seed` parameter allows users to create multiple margin accounts by using different seed values.
- The `features` parameter is converted to `AccountFeatureFlags` and controls account restrictions and behaviors.

```
seed: u16 // An arbitrary integer used to derive the new account address
features: u16 // Bit flags representing account feature configuration
```

Accounts

- **`owner`**: The owner of the new margin account.
  - Signer: Yes.
  - Init: No.
  - PDA: No.
  - Mutable: No.
  - Constraints: Must be the owner referenced in the permit account.
- **`permit`**: Permission account that enables the user to access resources within an airspace.
  - Signer: No.
  - Init: No.
  - PDA: Yes.
  - Mutable: No.
  - Constraints: Must have `owner` as its owner field via `has_one = owner` constraint.
- **`payer`**: Account that pays for the rent of the new margin account.
  - Signer: Yes.
  - Init: No.
  - PDA: No.
  - Mutable: Yes.
  - Constraints: Must have sufficient funds to pay for account creation.

- **`margin_account`**: The margin account being initialized.
    - Signer: No.
    - Init: Yes.
    - PDA: Yes (seeds: `[owner.key, permit.airspace, seed.to_le_bytes()]`).
    - Mutable: Yes.
    - Constraints: Cannot be reinitialized once created. Space allocated is `8 + std::mem::size_of::<MarginAccount>()`.
- **`system_program`**: Solana system program for account creation.
    - Signer: No.
    - Init: No.
    - PDA: No.
    - Mutable: No.
    - Constraints: Must be the system program.

Additional checks and behavior

- Validates that feature flags do not contain unknown bits.
- Prevents setting the `VIOLATION` flag during account creation.
- Ensures at most one feature flag bit is set (excluding combinations).
- The margin account is initialized with the airspace from the permit, owner key, seed, bump, and feature flags.
- Uses PDA derivation to ensure deterministic account addresses while allowing multiple accounts per user.
- Emits `AccountCreated` event on success.

CPI

- Calls the system program to create and initialize the margin account.

Instruction: `close_account`

This instruction closes a margin account and returns the rent to a receiver account. The account can only be closed if it has no open positions remaining.

Input parameters

This instruction takes no additional parameters beyond the accounts.

Accounts

- **`owner`**: The owner of the account being closed.
    - Signer: Yes.
    - Init: No.
    - PDA: No.
    - Mutable: No.
    - Constraints: Must be the owner of the margin account.
- **`receiver`**: Account that receives the returned rent from the closed margin account.
    - Signer: No.
    - Init: No.
    - PDA: No.
    - Mutable: Yes.
    - Constraints: No.
- **`margin_account`**: The margin account being closed.
    - Signer: No.
    - Init: No.
    - PDA: Yes.
    - Mutable: Yes.
    - Constraints: Must have `owner` as its owner.

Additional checks and behavior

- Verifies that the margin account has no open positions by checking `account.positions().count() > 0`.
- Returns `AccountNotEmpty` error if any positions remain.
- The account closure and rent transfer is handled automatically by Anchor's `close` constraint.
- Emits `AccountClosed` event upon successful closure.

CPI

No external CPI calls are made. Account closure is handled by the Anchor framework.

Instruction: `register_position`

This instruction registers a new position (token account) under the margin account's control for adapter-administered tokens. It creates a token account to hold the adapter-provided tokens, which represent a user's deposit with that adapter.

Input parameters

This instruction takes no additional parameters beyond the accounts.

Accounts

- **`authority`**: The authority that can change the margin account.
    - Signer: Yes.
    - Init: No.
    - PDA: No.
    - Mutable: No.
    - Constraints: Must be the verified authority for the margin account.
- **`payer`**: Account paying for rent costs.
    - Signer: Yes.
    - Init: No.
    - PDA: No.
    - Mutable: Yes.
    - Constraints: Must have sufficient funds for token-account creation.
- **`margin_account`**: The margin account to register the position with.
    - Signer: No.
    - Init: No.
    - PDA: Yes.
    - Mutable: Yes.
    - Constraints: Must have the same airspace as the token config.
- **`position_token_mint`**: The mint for the position token being registered.
    - Signer: No.
    - Init: No.
    - PDA: No.
    - Mutable: No.
    - Constraints: Must match the config's mint.
- **`config`**: The margin config for the token.
    - Signer: No.
    - Init: No.
    - PDA: Yes.
    - Mutable: No.
    - Constraints: Must be a valid token-configuration account for the airspace and mint.
- **`token_account`**: The token account to hold position assets in custody of the margin

account.

  - Signer: No.
  - Init: Yes.
  - PDA: Yes (seeds: `[margin_account.key, position_token_mint.key]`).
  - Mutable: Yes.
  - Constraints: Authority is set to the margin account, and the mint matches the position token mint.
- **`token_program`**: The SPL Token Program.
  - Signer: No.
  - Init: No.
  - PDA: No.
  - Mutable: No.
  - Constraints: Must be a token program.
- **`rent`**: The rent sysvar.
  - Signer: No.
  - Init: No.
  - PDA: No.
  - Mutable: No.
  - Constraints: Must be the rent sysvar.
- **`system_program`**: The system program for account creation.
  - Signer: No.
  - Init: No.
  - PDA: No.
  - Mutable: No.
  - Constraints: Must be the system program.

Additional checks and behavior

- Validates that the position does not already exist via `account.has_position(&position_token.key())`.
- Returns `PositionAlreadyExists` error if position is already registered.
- Checks that the config has an adapter program via `config.adapter_program().ok_or_else()`.
- Creates a `PositionConfigUpdate` from the token config with the adapter program.
- Registers position with `[Approver::MarginAccountAuthority]` approval.
- May fail if the account has reached a maximum number of positions (`MAX_USER_POSITIONS = 24`).

CPI

- Calls the SPL Token Program to create the token account.

Instruction: `update_position_balance`

This instruction manually updates the recorded balance of a specific position within a margin account to match the actual balance stored by the SPL Token Account. This is useful when tokens are deposited directly without invoking the margin program.

Input parameters

This instruction takes no additional parameters beyond the accounts.

Accounts

- **`margin_account`**: The margin account to update.
  - Signer: No.
  - Init: No.
  - PDA: Yes.
  - Mutable: Yes.
  - Constraints: Must contain a registered position for the provided token account.
- **`token_account`**: The token account to update the balance for.
  - Signer: No.
  - Init: No.
  - PDA: No (can be any token account).
  - Mutable: No.
  - Constraints: Must be a valid token account, and its mint and key must match a registered position.

Additional checks and behavior

- Permissionless instruction.
- Updates the position balance in the margin account to match the current token account balance.
- Uses the current unix timestamp from the syscall for balance-update timing.
- The `set_position_balance` function internally validates that the position exists for the given mint and token account key.

- If the position does not exist in the margin account, the function will fail.

CPI

No external CPI calls are made. This instruction only reads token account data and updates internal margin account state.

Instruction: `adapter_invoke`

This instruction allows the margin-account owner to execute instructions on external adapter programs, providing the margin account as a signer to grant authority over tokens held by the margin account.

Input parameters

The `instructions` parameter contains a list of instruction data structures that will be executed on adapter programs with the margin account as a signer.

```
instructions: Vec<IxData> // Vector of instruction data to pass to adapter
    program
```

Accounts

- **`owner`**: The authority that owns the margin account.
  - Signer: Yes.
  - Init: No.
  - PDA: No.
  - Mutable: No.
  - Constraints: Must be the owner of the margin account.
- **`margin_account`**: The margin account to proxy actions for.
  - Signer: No.
  - Init: No.
  - PDA: Yes.
  - Mutable: Yes.
  - Constraints: Must have `owner` as its owner and must not be undergoing liquidation.
- **Remaining accounts**: Variable number of accounts passed to adapter programs.
  - Signer: Varies.

- Init: Varies.
- PDA: Varies.
- Mutable: Varies.
- Constraints: No.

Additional checks and behavior

- Prevents execution if the margin account is being liquidated (`liquidator != Pubkey::default()`).
- Emits `AdapterInvokeBegin` and `AdapterInvokeEnd` events to mark invocation boundaries.
- Calls `adapter::invoke_many()` with the margin account as signer (`true` parameter).
- Performs health check via `valuation().verify_healthy()` after adapter execution.
- Validates position feature violations via `assert_position_feature_violation()`.
- The adapter validation is performed within `invoke_many()` using adapter metadata accounts.

CPI

- Makes CPI calls to validated adapter programs specified in the instructions.
- Adapter programs receive the margin account as a signer, granting them authority over margin account tokens.
- Each adapter program must have corresponding metadata configuration to be considered valid.

Instruction: `close_position`

This instruction closes out a position from a margin account, removing it from the account and optionally closing the associated token account. This instruction is used when a user no longer wants to maintain a specific position and needs to free up position slots for new positions.

Input parameters

This instruction takes no additional parameters beyond the accounts.

Accounts

- **`authority`**: The authority that can change the margin account.
  - Signer: Yes.
  - Init: No.

    - PDA: No.
    - Mutable: No.
    - Constraints: Must be the verified authority for the margin account.
- **`receiver`**: The receiver for the rent released from the closed token account.
    - Signer: No.
    - Init: No.
    - PDA: No.
    - Mutable: Yes.
    - Constraints: No.
- **`margin_account`**: The margin account with the position to close.
    - Signer: No.
    - Init: No.
    - PDA: Yes.
    - Mutable: Yes.
    - Constraints: Must have `authority` as its authority.
- **`position_token_mint`**: The mint for the position token being deregistered.
    - Signer: No.
    - Init: No.
    - PDA: No.
    - Mutable: No.
    - Constraints: Must match the mint of the position being closed.
- **`token_account`**: The token account for the position being closed.
    - Signer: No.
    - Init: No.
    - PDA: No.
    - Mutable: Yes.
    - Constraints: Must have a mint matching the `position_token_mint` and be associated with the correct token program.
- **`token_program`**: The SPL Token Program interface.
    - Signer: No.
    - Init: No.
    - PDA: No.
    - Mutable: No.
    - Constraints: Must be a token program.

Additional checks and behavior

- Verifies the authority has permission to modify the margin account via

`account.verify_authority()`.

- Calls `account.unregister_position()` with `Approver::MarginAccountAuthority` to remove the position from the margin account.
- The position must have zero balance to be closed (checked in `unregister_position`).
- If the token account is owned by the margin account, it closes the token account via CPI to the token program.
- The margin account acts as the authority for closing its own token accounts using its signer seeds.
- Rent from the closed token account is transferred to the specified receiver.

CPI

- Calls `token_2022::close_account` if the token account is owned by the margin account.
- Uses the margin account's signer seeds to authorize the token-account closure.

Instruction: `verify_healthy`

This instruction verifies that a margin account is healthy by validating that the collateralization ratio is above the minimum required threshold.

Input parameters

This instruction takes no additional parameters beyond the accounts.

Accounts

- **`margin_account`**: The account to verify the health of.
  - Signer: No.
  - Init: No.
  - PDA: Yes.
  - Mutable: No.
  - Constraints: Must be an initialized margin account.

Additional checks and behavior

- Loads the margin-account data.
- Calculates the account valuation using the current timestamp from `sys().unix_timestamp()`.
- Calls `valuation.verify_healthy()`, which checks if the account meets minimum collateral requirements.

- Emits a `VerifiedHealthy` event upon successful verification.
- If the account is unhealthy, the instruction will fail with an appropriate error.
- The health check considers all positions, their values, and collateral requirements.

CPI

No external CPI calls are made.

Instruction: `verify_unhealthy`

This instruction verifies that a margin account is unhealthy by validating that the collateralization ratio is below the minimum required threshold.

Input parameters

This instruction takes no additional parameters beyond the accounts.

Accounts

- **`margin_account`**: The account to verify the health of.
  - Signer: No.
  - Init: No.
  - PDA: Yes.
  - Mutable: No.
  - Constraints: Must be an initialized margin account.

Additional checks and behavior

- Loads the margin-account data.
- Calculates the account valuation using the current timestamp from `sys().unix_timestamp()`.
- Calls `valuation.verify_unhealthy()`, which checks if the account fails to meet minimum collateralization requirements.
- Emits a `VerifiedUnhealthy` event upon successful verification of unhealthy status.
- If the account is actually healthy, the instruction will fail with an appropriate error.
- The health check considers all positions, their values, and collateral requirements.

CPI

No external CPI calls are made. This is a read-only verification instruction.

Instruction: `accounting_invoke`

This instruction performs actions by invoking other programs for the purpose of refreshing the state of a margin account to be consistent with actual underlying prices or positions. Unlike `adapter_invoke`, this instruction does not provide the margin account as a signer, preventing adapters from modifying token balances and restricting them to read-only operations for accounting updates.

Input parameters

The `instructions` parameter contains the instruction data that will be passed to the invoked adapter programs for execution.

```
instructions: Vec<IxData> // Vector of instruction data to pass to adapter
    programs
```

Accounts

- **`margin_account`**: The margin account to proxy an action for.
  - Signer: No.
  - Init: No.
  - PDA: Yes.
  - Mutable: Yes.
  - Constraints: No.
- **Remaining accounts**: Variable number of accounts passed to adapter programs.
  - Signer: Varies.
  - Init: Varies.
  - PDA: Varies.
  - Mutable: Varies.
  - Constraints: No.

Additional checks and behavior

- Emits `AccountingInvokeBegin` event at the start of processing.

- Calls `adapter::invoke_many()` with `is_signer = false`, meaning the margin account is not provided as a signer to invoked programs.
- Adapters can only perform read-only operations and report position-state changes.
- This is a permissionless way of updating position values that require adapter-provided updates.
- Emits `AccountingInvokeEnd` event upon completion.
- All position changes reported by adapters are processed and may emit `PositionEvent` events.

CPI

- Calls multiple adapter programs via `adapter::invoke_many()`.
- The margin account does **not** act as a signer in these CPIs, restricting adapters to read-only operations.
- Adapters cannot transfer tokens or modify balances, only report state changes.

Instruction: `liquidate_begin`

The instruction begins the liquidation process for an unhealthy margin account. This instruction creates a liquidation state account and sets the margin account into liquidation mode, preventing the owner from taking actions until the liquidation is completed or timed out.

Input parameters

This instruction takes no additional parameters beyond the accounts.

Accounts

- **`margin_account`**: The account in need of liquidation.
  - Signer: No.
  - Init: No.
  - PDA: Yes.
  - Mutable: Yes.
  - Constraints: Must be unhealthy (failing collateralization requirements).
- **`payer`**: The address paying rent for the liquidation state account.
  - Signer: Yes.
  - Init: No.
  - PDA: No.
  - Mutable: Yes.

    - Constraints: Must have sufficient funds to create the liquidation account.
- **`liquidator`**: The liquidator account performing the liquidation actions.
    - Signer: Yes.
    - Init: No.
    - PDA: No.
    - Mutable: No.
    - Constraints: Must have a valid liquidation permit.
- **`permit`**: The permit allowing the liquidator to perform liquidations.
    - Signer: No.
    - Init: No.
    - PDA: Yes.
    - Mutable: No.
    - Constraints: Must be owned by the liquidator, contain `LIQUIDATE` permissions, and be in the same airspace as the margin account.
- **`liquidation`**: Account to persist the state of the liquidation.
    - Signer: No.
    - Init: Yes.
    - PDA: Yes (seeds: `["liquidation", margin_account, liquidator]`).
    - Mutable: Yes.
    - Constraints: No — account must exist at the address derived from the seeds.
- **`system_program`**: The Solana system program for account creation.
    - Signer: No.
    - Init: No.
    - PDA: No.
    - Mutable: No.
    - Constraints: Must be the system program.

Additional checks and behavior

- Verifies the margin account is unhealthy via `valuation.verify_unhealthy()`.
- Checks if the account is already being liquidated and handles it accordingly:
    - If already claimed by the same liquidator, returns successfully.
    - If claimed by a different liquidator, returns `Liquidating` error.
    - If not being liquidated, claims it for the current liquidator.
- Calculates maximum equity loss allowed based on account liabilities using the formula `M * liabilities + B` where M and B are protocol constants.
- Calculates maximum available collateral increase limit as a percentage of required collateral.

- Creates liquidation state with timestamp, max equity loss, and collateral limits.
- Emits `LiquidationBegun` event with comprehensive liquidation details.

CPI

- Calls the system program to create the liquidation-state account.

Instruction: `liquidate_end`

This instruction ends the liquidation state for a margin account, allowing the account owner to regain control. This instruction can be called by the liquidator who initiated the liquidation or by anyone after the liquidation time-out period has elapsed.

Input parameters

This instruction takes no additional parameters beyond the accounts.

Accounts

- **`authority`**: The pubkey calling the instruction to end liquidation.
  - Signer: Yes.
  - Init: No.
  - PDA: No.
  - Mutable: Yes.
  - Constraints: Must be either the liquidator (before time-out) or any signer (after time-out).
- **`margin_account`**: The account that was under liquidation.
  - Signer: No.
  - Init: No.
  - PDA: Yes.
  - Mutable: Yes.
  - Constraints: Must have a liquidator matching the liquidation state.
- **`liquidation`**: Account storing the liquidation state.
  - Signer: No.
  - Init: No.
  - PDA: Yes.
  - Mutable: Yes.
  - Constraints: Must reference the correct margin account.

Additional checks and behavior

- Checks if the liquidation has timed out by comparing the current timestamp with start time plus `LIQUIDATION_TIMEOUT` (60 seconds).
- Regarding the authorization logic — before time-out, only the original liquidator can end the liquidation, and after time-out, anyone can end the liquidation (emergency mechanism).
- Calls `account.end_liquidation()` to reset the margin account's liquidator field.
- The liquidation account is automatically closed via Anchor's `close` constraint.
- Emits the `LiquidationEnded` event with details about who ended it and whether it timed out.

CPI

No external CPI calls are made. Account closure is handled by Anchor's `close` constraint.

Instruction: `liquidator_invoke`

This instruction performs actions by invoking adapter programs during the liquidation process. This instruction provides the margin account as a signer to adapter programs, granting them authority to modify token balances for liquidation purposes while enforcing strict equity loss and value-extraction limits.

Input parameters

- The `instructions` parameter contains the instruction data that will be passed to the invoked adapter programs for liquidation actions.

```
instructions: Vec<IxData> // Vector of instruction data to pass to adapter
    programs
```

Accounts

- **`liquidator`**: The liquidator processing the margin account.
  - Signer: Yes.
  - Init: No.
  - PDA: No.
  - Mutable: No.
  - Constraints: Must match the liquidator in the liquidation state.
- **`liquidation`**: Account storing the liquidation state.

  - Signer: No.
  - Init: No.
  - PDA: Yes.
  - Mutable: Yes.
  - Constraints: Must reference the correct liquidator and margin account. Must not be in fee-collection mode.
- **`margin_account`**: The margin account being liquidated.
  - Signer: No.
  - Init: No.
  - PDA: Yes.
  - Mutable: Yes.
  - Constraints: Must match the margin account in the liquidation state.
- **`liquidator_fee_mint`**: The mint in which the liquidator will accrue fees.
  - Signer: No.
  - Init: No.
  - PDA: No.
  - Mutable: No.
  - Constraints: No.
- **`liquidator_fee_token_program`**: The token program for the fee mint.
  - Signer: No.
  - Init: No.
  - PDA: No.
  - Mutable: No.
  - Constraints: Must be a token program.
- **Remaining accounts**: Variable number of accounts passed to adapter programs.
  - Signer: Varies.
  - Init: Varies.
  - PDA: Varies.
  - Mutable: Varies.
  - Constraints: No.

Additional checks and behavior

- Records starting-account valuation before adapter invocations.
- Emits `LiquidatorInvokeBegin` event.
- Calls `adapter::invoke_many()` with `is_signer = true`, providing the margin account as a signer.
- Calculates liquidation fees based on token-balance changes:

    - Filters changes to the fee mint for relevant operations (borrow, repay, external transfers)
    - Calculates fee-eligible tokens as the minimum of increases and repayments
    - Applies the liquidation fee rate using the formula `fee_rate / (1 + fee_rate) * eligible_amount`
- Accrues liquidation fees in the liquidation state.
- Updates and verifies liquidation constraints:
    - Tracks equity loss and ensures it does not exceed maximum allowed
    - Monitors collateral changes and prevents reductions
    - Enforces available collateral limits
- Emits `LiquidatorInvokeEnd` event with liquidation summary.

CPI

- Calls multiple adapter programs via `adapter::invoke_many()` with the margin account as signer.
- Adapters can transfer tokens and modify balances under liquidation constraints.

Instruction: `refresh_position_config`

This instruction updates the configuration metadata for a position in a margin account when the token configuration has changed after the position was created. This instruction allows authorized refreshers to update position parameters to match updated token configurations.

Input parameters

This instruction takes no additional parameters beyond the accounts.

Accounts

- **`margin_account`**: The margin account with the position to be refreshed.
    - Signer: No.
    - Init: No.
    - PDA: Yes.
    - Mutable: Yes.
    - Constraints: Must be in the same airspace as the token config via the airspace constraint.
- **`config`**: The updated config account for the token.
    - Signer: No.

    - Init: No.
    - PDA: Yes.
    - Mutable: No.
    - Constraints: Must be in the same airspace as the margin account via the `WrongAirspace` check.
- **`permit`**: Permit that authorizes the refresher.
    - Signer: No.
    - Init: No.
    - PDA: Yes.
    - Mutable: No.
    - Constraints: Must be in the same airspace as the margin account via `WrongAirspace` check.
- **`refresher`**: Account that is authorized to refresh position metadata.
    - Signer: Yes.
    - Init: No.
    - PDA: No.
    - Mutable: No.
    - Constraints: Must have valid permit with `REFRESH_POSITION_CONFIG` permissions.

Additional checks and behavior

- Validates the refresher's permit contains `REFRESH_POSITION_CONFIG` permissions via `permit.validate()`.
- Calls `account.refresh_position_metadata()` with updated configuration parameters: token mint, token kind, value modifier, max staleness, and token features.
- There is special handling for restricted tokens:
    - If a token becomes restricted and the margin account has no feature flags, sets the `VIOLATION` flag.
    - This mechanism enforces compliance when token configurations change to restricted status.
    - Users must close restricted positions to remedy violations.
- Updates position metadata in place without requiring position closure and recreation.

CPI

No external CPI calls are made. This is a metadata update operation.

Instruction: `refresh_deposit_position`

This instruction refreshes the price and optionally the balance for a deposit position in a margin account. This instruction updates position data using current oracle prices and actual token account balances to ensure accurate valuation.

Input parameters

This instruction takes no additional parameters beyond the accounts.

Accounts

- **`margin_account`**: The account to update.
    - Signer: No.
    - Init: No.
    - PDA: Yes.
    - Mutable: Yes.
    - Constraints: No.
- **`config`**: The margin config for the token.
    - Signer: No.
    - Init: No.
    - PDA: Yes.
    - Mutable: No.
    - Constraints: Must be in the same airspace as the margin account.
- **`price_oracle`**: Oracle account for token pricing.
    - Signer: No.
    - Init: No.
    - PDA: No.
    - Mutable: No.
    - Constraints: Must be a valid Pyth Pull Receiver program account.
- **`redemption_quote_oracle`**: Optional oracle for quote token when using redemption rates.
    - Signer: No.
    - Init: No.
    - PDA: No.
    - Mutable: No.
    - Constraints: Must be a valid Pyth Pull Receiver program account, if provided.

Optional account via `remaining_accounts`:

- **`position_token_account`**: Token account to refresh balance for (first remaining account if provided).

Additional checks and behavior

- Extracts oracle configuration from the token config and validates it exists.
- Constructs `PriceChangeInfo` from oracle accounts using `try_from_oracle_accounts()`.
- If a position token account is provided in the remaining accounts, it
  - reads the current token balance using `token::accessor::amount()`, and
  - updates the position balance in the margin account with current timestamp.
- Updates the position price using the oracle-derived price information.
- Both price and balance updates use the current system timestamp for staleness tracking.
- Oracle verification ensures only authorized price feeds can update position values.

CPI

No external CPI calls are made. This instruction reads oracle data and updates margin-account state.

Instruction: `create_deposit_position`

This instruction creates a new position for holding SPL token deposits directly by a margin account. This instruction registers an associated token account with the margin account to enable direct token deposits and withdrawals.

Input parameters

This instruction takes no additional parameters beyond the accounts.

Accounts

- **`authority`**: The authority that can change the margin account.
  - Signer: Yes.
  - Init: No.
  - PDA: No.
  - Mutable: No.
  - Constraints: Must be the verified authority for the margin account.

- **`payer`**: The address paying for rent of the associated token account.
    - Signer: Yes.
    - Init: No.
    - PDA: No.
    - Mutable: Yes.
    - Constraints: Must have sufficient funds to create the associated token account.
- **`margin_account`**: The margin account to register this deposit account with.
    - Signer: No.
    - Init: No.
    - PDA: Yes.
    - Mutable: Yes.
    - Constraints: No.
- **`mint`**: The mint for the token being stored in this deposit account.
    - Signer: No.
    - Init: No.
    - PDA: No.
    - Mutable: No.
    - Constraints: Must match the mint in the token configuration.
- **`config`**: The margin config for the token.
    - Signer: No.
    - Init: No.
    - PDA: Yes.
    - Mutable: No.
    - Constraints: Must be in the same airspace as the margin account and have the correct mint.
- **`token_account`**: The associated token account to store deposits.
    - Signer: No.
    - Init: No.
    - PDA: Yes (associated token account).
    - Mutable: No.
    - Constraints: Must be the associated token account for the mint and margin account.
- **`associated_token_program`**: The associated token program.
    - Signer: No.
    - Init: No.
    - PDA: No.
    - Mutable: No.

    - Constraints: Must be the associated token program.
- **`mint_token_program`**: The token program for the mint.
    - Signer: No.
    - Init: No.
    - PDA: No.
    - Mutable: No.
    - Constraints: Must be a token program.
- **`rent`**: The rent sysvar.
    - Signer: No.
    - Init: No.
    - PDA: No.
    - Mutable: No.
    - Constraints: No.
- **`system_program`**: The system program.
    - Signer: No.
    - Init: No.
    - PDA: No.
    - Mutable: No.
    - Constraints: Must be the system program.

Additional checks and behavior

- Verifies the authority has permission to modify the margin account.
- Creates a `PositionConfigUpdate` from the token configuration with token mint, decimals, token account address, and adapter program (if any).
- Registers the new position with `Approver::MarginAccountAuthority`.
- The associated token account is derived deterministically from the margin account and mint.
- Position registration may fail if the account has reached maximum positions.

CPI

The associated-token-account creation is handled by Anchor's constraint system, which may involve CPI calls to create the account if it does not exist.

Instruction: `transfer_deposit`

This instruction transfers tokens into or out of a token account being used for deposits by a margin account. This instruction handles both deposits and withdrawals while maintaining accurate

position balance tracking.

Input parameters

The `amount` parameter specifies how many tokens to transfer between the source and destination accounts.

```
amount: u64 // The amount of tokens to transfer
```

Accounts

- **`owner`**: The authority that owns the margin account.
  - Signer: Yes.
  - Init: No.
  - PDA: No.
  - Mutable: No.
  - Constraints: Must be the owner of the margin account.
- **`margin_account`**: The margin account associated with the deposit account.
  - Signer: No.
  - Init: No.
  - PDA: Yes.
  - Mutable: Yes.
  - Constraints: Must have `owner` as its owner.
- **`source_owner`**: The authority for the source account.
  - Signer: No.
  - Init: No.
  - PDA: No.
  - Mutable: No.
  - Constraints: Must be the authority for the `source` token account.
- **`source`**: The source account to transfer tokens from.
  - Signer: No.
  - Init: No.
  - PDA: No.
  - Mutable: Yes.
  - Constraints: Must have a matching mint and token program.
- **`destination`**: The destination account to transfer tokens to.
  - Signer: No.

    - Init: No.
    - PDA: No.
    - Mutable: Yes.
    - Constraints: Must have a matching mint and token program.
- **`mint`**: The mint for the tokens being transferred.
    - Signer: No.
    - Init: No.
    - PDA: No.
    - Mutable: No.
    - Constraints: Must match the mint of both source and destination accounts.
- **`token_program`**: The token program interface.
    - Signer: No.
    - Init: No.
    - PDA: No.
    - Mutable: No.
    - Constraints: Must be a token program.

Additional checks and behavior

- Verifies the token mint is registered as a position in the margin account.
- Handles two transfer scenarios:
    1. Margin account as source — uses margin account signer seeds for authorization.
    2. External source — uses the provided `source_owner` as authority.
- Executes `transfer_checked` CPI with appropriate authority and signer seeds.
- Reloads the affected token account after transfer to get updated balance.
- Updates the margin account's position balance with the new token amount and current timestamp.
- The instruction is flexible for both deposits (external → margin account) and withdrawals (margin account → external).

CPI

- Calls `token_interface::transfer_checked` with either
    - the margin account as signer authority (when transferring from margin account positions), or
    - the external source owner as authority (when transferring from external accounts).

Instruction: `configure_token`

This instruction sets the configuration for a token, which allows it to be used as a position in a margin account. This instruction creates or updates token configuration within an airspace, defining how the token can be used for collateral, claims, and other margin operations.

Input parameters

If `update` is `Some(TokenConfigUpdate)`, the configuration is created or updated with the provided parameters. If `None`, the configuration account is deleted and rent is returned.

The `TokenConfigUpdate` struct contains the following.

- `underlying_mint`: The underlying token represented.
- `underlying_mint_token_program`: The underlying token's program.
- `admin`: Administration authority (Adapter or Margin).
- `token_kind`: Description of token type (`Collateral`, `AdapterCollateral`, `Claim`).
- `value_modifier`: Modifier to adjust token value based on kind.
- `max_staleness`: Maximum acceptable staleness for token balances (seconds).
- `token_features`: Feature flags including restrictions.

```
update: Option<TokenConfigUpdate> // The configuration update to apply, or
    None to delete
```

Accounts

- **`authority`**: The authority allowed to make configuration changes.
  - Signer: Yes.
  - Init: No.
  - PDA: No.
  - Mutable: No.
  - Constraints: Must be the airspace authority.
- **`airspace`**: The airspace being modified.
  - Signer: No.
  - Init: No.
  - PDA: Yes.
  - Mutable: No.
  - Constraints: Authority must match the airspace authority.
- **`payer`**: The payer for rent costs when creating configuration.
  - Signer: Yes.

    - Init: No.
    - PDA: No.
    - Mutable: Yes.
    - Constraints: Must have sufficient funds for account creation.
- **`mint`**: The mint for the token being configured.
    - Signer: No.
    - Init: No.
    - PDA: No.
    - Mutable: No.
    - Constraints: Must be a token mint.
- **`token_config`**: The config account to be modified.
    - Signer: No.
    - Init: Yes (if needed).
    - PDA: Yes (seeds: `["token_config", airspace, mint]`).
    - Mutable: Yes.
    - Constraints: Must have a size of `8 + std::mem::size_of::<TokenConfig>()` bytes.
- **`system_program`**: The system program for account operations.
    - Signer: No.
    - Init: No.
    - PDA: No.
    - Mutable: No.
    - Constraints: Must be the system program.

Additional checks and behavior

- Emits the `TokenConfigured` event before processing.
- For deletion (`None` update), closes the account and returns rent to payer.
- Validates configuration parameters:
    - Token-balance staleness cannot exceed `MAX_TOKEN_STALENESS`.
    - Token features must be known and valid combinations.
    - A restricted flag cannot be the only feature set.
    - At most, one nonrestricted feature flag can be set.
- Enforces immutability rules for existing configurations:
    - Token kind cannot be changed after creation.
    - Admin type transitions are restricted (Adapter <-> Margin not allowed).
    - Adapter addresses are immutable once set.
    - Feature flags cannot be changed once set (except `RESTRICTED`).

    - Underlying mint cannot be changed.
- Validates value modifier limits based on token kind.
- Updates all configuration fields and calls `config.validate()` for final verification.

CPI

- May call the system program for account closure via Anchor's `close` functionality.

Instruction: `configure_adapter`

This instruction sets the configuration for an adapter program, determining whether a program is authorized to be invoked by margin accounts within a specific airspace. This instruction controls which external programs can interact with margin accounts.

Input parameters

If `is_adapter` is `true`, the program is authorized as an adapter. If `false`, the configuration is deleted and the program is deauthorized.

```
is_adapter: bool // Whether to authorize the program as an adapter (true) or
    remove authorization (false)
```

Accounts

- **`authority`**: The authority allowed to make configuration changes.
    - Signer: Yes.
    - Init: No.
    - PDA: No.
    - Mutable: No.
    - Constraints: Must be the airspace authority.
- **`airspace`**: The airspace being modified.
    - Signer: No.
    - Init: No.
    - PDA: Yes.
    - Mutable: No.
    - Constraints: Authority must match the airspace authority.
- **`payer`**: The payer for rent costs when creating configuration.
    - Signer: Yes.

  - Init: No.
  - PDA: No.
  - Mutable: Yes.
  - Constraints: Must have sufficient funds for account creation.
- **`adapter_program`**: The adapter program being configured.
  - Signer: No.
  - Init: No.
  - PDA: No.
  - Mutable: No.
  - Constraints: No.
- **`adapter_config`**: The config account to be modified.
  - Signer: No.
  - Init: Yes (if needed).
  - PDA: Yes (seeds: `["adapter_config", airspace, adapter_program]`).
  - Mutable: Yes.
  - Constraints: Must have a size of `8 + std::mem::size_of::<AdapterConfig>()` bytes.
- **`system_program`**: The system program for account operations.
  - Signer: No.
  - Init: No.
  - PDA: No.
  - Mutable: No.
  - Constraints: Must be the system program.

Additional checks and behavior

- Emits the `AdapterConfigured` event with airspace, adapter program, and authorization status.
- For deauthorization (`is_adapter = false`), closes the account and returns rent to payer.
- For authorization (`is_adapter = true`), sets `adapter_program` field to the program's public key and sets the `airspace` field to the airspace's public key.
- The configuration serves as authorization for the margin program to invoke the adapter.
- Only programs with valid adapter configurations can be called via `adapter_invoke` or `liquidator_invoke`.

CPI

- May call the system program for account closure via Anchor's `close` functionality when deauthorizing.

Instruction: `configure_liquidator`

This instruction sets the liquidation permissions for an account within a specific airspace, determining whether the account is authorized to perform liquidations. This instruction is a wrapper around the general permit configuration system.

Input parameters

If `is_liquidator` is `true`, the `LIQUIDATE` permission is granted. If `false`, the permission is revoked.

```
is_liquidator: bool // Whether to grant liquidation permissions (true) or
    revoke them (false)
```

Accounts

- **`authority`**: The authority allowed to make configuration changes.
  - Signer: Yes.
  - Init: No.
  - PDA: No.
  - Mutable: No.
  - Constraints: Must be the airspace authority.
- **`airspace`**: The airspace being modified.
  - Signer: No.
  - Init: No.
  - PDA: Yes.
  - Mutable: No.
  - Constraints: Authority must match the airspace authority.
- **`payer`**: The payer for rent costs when creating permits.
  - Signer: Yes.
  - Init: No.
  - PDA: No.
  - Mutable: Yes.
  - Constraints: Must have sufficient funds for account creation.
- **`owner`**: The account being granted or revoked liquidation permissions.
  - Signer: No.
  - Init: No.
  - PDA: No.
  - Mutable: No.
  - Constraints: No.

- **`permit`**: The permit account storing permissions.
    - Signer: No.
    - Init: Yes (if needed).
    - PDA: Yes (seeds: `["permit", airspace, owner]`).
    - Mutable: Yes.
    - Constraints: Must have a size of `8 + std::mem::size_of::<Permit>()` bytes.
- **`system_program`**: The system program for account operations.
    - Signer: No.
    - Init: No.
    - PDA: No.
    - Mutable: No.
    - Constraints: Must be the system program.

Additional checks and behavior

- Calls the generic `configure_permit` function with the `Permissions::LIQUIDATE` flag.
- The permit system manages permissions using bitflags, allowing multiple permissions per account.
- If granting permissions, the `LIQUIDATE` flag is added to existing permissions.
- If revoking permissions, the `LIQUIDATE` flag is removed from existing permissions.
- If all permissions are removed, the permit account is closed and rent is returned.
- Emits the `PermitConfigured` event with updated permission set.

CPI

- May call the system program for account closure via Anchor's `close` functionality when all permissions are revoked.

Instruction: `configure_position_config_refresher`

This instruction sets the position-configuration refresh permissions for an account within a specific airspace, determining whether the account is authorized to refresh position configurations when token configurations change.

Input parameters

If `may_refresh` is `true`, the `REFRESH_POSITION_CONFIG` permission is granted. If `false`, the permission is revoked.

```
may_refresh: bool // Whether to grant refresh permissions (true) or revoke
    them (false)
```

Accounts

- **`authority`**: The authority allowed to make configuration changes.
    - Signer: Yes.
    - Init: No.
    - PDA: No.
    - Mutable: No.
    - Constraints: Must be the airspace authority.
- **`airspace`**: The airspace being modified.
    - Signer: No.
    - Init: No.
    - PDA: Yes.
    - Mutable: No.
    - Constraints: Authority must match the airspace authority.
- **`payer`**: The payer for rent costs when creating permits.
    - Signer: Yes.
    - Init: No.
    - PDA: No.
    - Mutable: Yes.
    - Constraints: Must have sufficient funds for account creation.
- **`owner`**: The account being granted or revoked refresh permissions.
    - Signer: No.
    - Init: No.
    - PDA: No.
    - Mutable: No.
    - Constraints: No.
- **`permit`**: The permit account storing permissions.
    - Signer: No.
    - Init: Yes (if needed).
    - PDA: Yes (seeds: `["permit", airspace, owner]`).
    - Mutable: Yes.
    - Constraints: Must have a size of `8 + std::mem::size_of::<Permit>()` bytes.
- **`system_program`**: The system program for account operations.

- Signer: No.
- Init: No.
- PDA: No.
- Mutable: No.
- Constraints: Must be the system program.

Additional checks and behavior

- Calls the generic `configure_permit` function with the `Permissions::REFRESH_POSITION_CONFIG` flag.
- The permit system manages permissions using bitflags, allowing multiple permissions per account.
- If granting permissions, the `REFRESH_POSITION_CONFIG` flag is added to existing permissions.
- If revoking permissions, the `REFRESH_POSITION_CONFIG` flag is removed from existing permissions.
- If all permissions are removed, the permit account is closed and rent is returned.
- Emits `PermitConfigured` event with updated permission set.

CPI

- May call the system program for account closure via Anchor's `close` functionality when all permissions are revoked.

Instruction: `admin_transfer_position`

This instruction allows the protocol governor to transfer any position from one margin account to another. This instruction is provided as an emergency mechanism to manually fix issues that occur in the protocol due to problematic user assets or other exceptional circumstances.

Input parameters

The `amount` parameter specifies how many tokens to transfer from the source to the target margin account.

```
amount: u64 // The amount of tokens to transfer between margin accounts
```

Accounts

- **`authority`**: The administrative authority.

  - Signer: Yes.
  - Init: No.
  - PDA: No.
  - Mutable: No.
  - Constraints: Must be the protocol governor address.
- **`target_account`**: The target margin account to move a position into.
  - Signer: No.
  - Init: No.
  - PDA: Yes.
  - Mutable: Yes.
  - Constraints: No.
- **`source_account`**: The source margin account to move a position out of.
  - Signer: No.
  - Init: No.
  - PDA: Yes.
  - Mutable: Yes.
  - Constraints: No.
- **`source_token_account`**: The token account to be moved from.
  - Signer: No.
  - Init: No.
  - PDA: No.
  - Mutable: Yes.
  - Constraints: Must match the token mint and token program and be owned by the source margin account.
- **`target_token_account`**: The token account to be moved into.
  - Signer: No.
  - Init: No.
  - PDA: No.
  - Mutable: Yes.
  - Constraints: Must match the token mint.
- **`token_mint`**: The mint for the tokens being transferred.
  - Signer: No.
  - Init: No.
  - PDA: No.
  - Mutable: No.
  - Constraints: Must match both token accounts.
- **`token_program`**: The token program interface.
  - Signer: No.

- Init: No.
- PDA: No.
- Mutable: No.
- Constraints: Must be a token program.

Additional checks and behavior

- Uses the source margin account's signer seeds to authorize the token transfer.
- Executes the `transfer_checked` CPI with the source margin account as authority.
- Reloads both token accounts after transfer to get updated balances.
- Updates both margin accounts' position balances with new token amounts and current timestamp.
- Emits the `TransferPosition` event with details about the transfer.

CPI

- Calls `token_interface::transfer_checked` with the source margin account as signer authority using its signer seeds.

Instruction: `init_lookup_registry`

This instruction creates a lookup-table registry account owned by a margin account. The registry serves as a container for address lookup tables that store addresses for accounts owned by the margin account, such as PDAs, pool accounts, and other accounts from adapters.

Input parameters

This instruction takes no additional parameters beyond the accounts.

Accounts

- **`authority`**: The authority that can register a lookup table for a margin account.
  - Signer: Yes.
  - Init: No.
  - PDA: No.
  - Mutable: No.
  - Constraints: Must be the verified authority for the margin account.
- **`payer`**: The payer of the transaction.
  - Signer: Yes.

    - Init: No.
    - PDA: No.
    - Mutable: Yes.
    - Constraints: Must have sufficient funds for account creation.
- **`margin_account`**: The margin account to create this lookup registry for.
    - Signer: No.
    - Init: No.
    - PDA: Yes.
    - Mutable: No.
    - Constraints: No.
- **`registry_account`**: The registry account being created.
    - Signer: No.
    - Init: No.
    - PDA: No.
    - Mutable: Yes.
    - Constraints: No.
- **`registry_program`**: The lookup-table registry program.
    - Signer: No.
    - Init: No.
    - PDA: No.
    - Mutable: No.
    - Constraints: Must be the lookup-table registry program.
- **`system_program`**: The system program for account creation.
    - Signer: No.
    - Init: No.
    - PDA: No.
    - Mutable: No.
    - Constraints: Must be the system program.

Additional checks and behavior

- Verifies the authority has permission to modify the margin account.
- Uses the margin account's signer seeds to act as authority for the registry creation.
- Makes a CPI call to the lookup-table registry program to initialize the registry account.

CPI

- Calls `lookup_table_registry::cpi::init_registry_account` with the margin

account as authority using its signer seeds.

Instruction: `create_lookup_table`

This instruction creates a lookup table in a registry account owned by a margin account. This instruction enables the creation of address lookup tables that can store frequently used account addresses to optimize transaction size and composition.

Input parameters

The `recent_slot` parameter must be a recent slot number as required by the Solana address lookup-table program. The `discriminator` provides uniqueness for multiple lookup tables within the same registry.

```
recent_slot: u64,      // A recent slot number for lookup table initialization
discriminator: u64     // A unique discriminator for the lookup table
```

Accounts

- **`authority`**: The authority that can register a lookup table for a margin account.
  - Signer: Yes.
  - Init: No.
  - PDA: No.
  - Mutable: No.
  - Constraints: Must be the verified authority for the margin account.
- **`payer`**: The payer of the transaction.
  - Signer: Yes.
  - Init: No.
  - PDA: No.
  - Mutable: Yes.
  - Constraints: Must have sufficient funds for account creation.
- **`margin_account`**: The margin account that owns the registry.
  - Signer: No.
  - Init: No.
  - PDA: Yes.
  - Mutable: No.
  - Constraints: No.
- **`registry_account`**: The registry account that will contain the lookup table.

    - Signer: No.
    - Init: No.
    - PDA: No.
    - Mutable: Yes.
    - Constraints: Must be owned by the margin account and managed by the registry program.
- **`lookup_table`**: The lookup table being created.
    - Signer: No.
    - Init: No.
    - PDA: No.
    - Mutable: Yes.
    - Constraints: No.
- **`address_lookup_table_program`**: The Solana Address Lookup Table program.
    - Signer: No.
    - Init: No.
    - PDA: No.
    - Mutable: No.
    - Constraints: Must be the Address Lookup Table program.
- **`registry_program`**: The lookup-table registry program.
    - Signer: No.
    - Init: No.
    - PDA: No.
    - Mutable: No.
    - Constraints: Must be the lookup-table registry program.
- **`system_program`**: The system program for account creation.
    - Signer: No.
    - Init: No.
    - PDA: No.
    - Mutable: No.
    - Constraints: Must be the system program.

Additional checks and behavior

- Verifies the authority has permission to modify the margin account.
- Uses the margin account's signer seeds to act as authority for the lookup-table creation.
- Makes a CPI call to the lookup-table registry program to create the lookup table.

CPI

- Calls `lookup_table_registry::cpi::create_lookup_table` with the margin account as authority using its signer seeds.

Instruction: `append_to_lookup`

This instruction appends addresses to an existing lookup table in a registry account owned by a margin account. This instruction allows adding frequently used account addresses to optimize transaction composition and reduce transaction size.

Input parameters

The `addresses` parameter contains the list of account addresses to be added to the lookup table.

```
addresses: Vec<Pubkey> // Vector of public keys to append to the lookup table
```

Accounts

- **`authority`**: The authority that can manage lookup tables for a margin account.
  - Signer: Yes.
  - Init: No.
  - PDA: No.
  - Mutable: No.
  - Constraints: Must be the verified authority for the margin account.
- **`payer`**: The payer of the transaction.
  - Signer: Yes.
  - Init: No.
  - PDA: No.
  - Mutable: Yes.
  - Constraints: Must have sufficient funds for account expansion.
- **`margin_account`**: The margin account that owns the registry and lookup table.
  - Signer: No.
  - Init: No.
  - PDA: Yes.
  - Mutable: No.
  - Constraints: No.
- **`registry_account`**: The registry account containing the lookup table.

    - Signer: No.
    - Init: No.
    - PDA: No.
    - Mutable: Yes.
    - Constraints: Must be owned by the margin account and managed by the registry program.
- **`lookup_table`**: The lookup table being modified.
    - Signer: No.
    - Init: No.
    - PDA: No.
    - Mutable: Yes.
    - Constraints: Must exist within the registry and be owned by the margin account.
- **`address_lookup_table_program`**: The Solana Address Lookup Table program.
    - Signer: No.
    - Init: No.
    - PDA: No.
    - Mutable: No.
    - Constraints: Must be the Address Lookup Table program.
- **`registry_program`**: The lookup-table registry program.
    - Signer: No.
    - Init: No.
    - PDA: No.
    - Mutable: No.
    - Constraints: Must be the lookup-table registry program.
- **`system_program`**: The system program for account operations.
    - Signer: No.
    - Init: No.
    - PDA: No.
    - Mutable: No.
    - Constraints: Must be the system program.

Additional checks and behavior

- Verifies the authority has permission to modify the margin account.
- Uses the margin account's signer seeds to act as authority for the lookup-table operations.
- Makes a CPI call to the lookup-table registry program to append addresses.

CPI

- Calls `lookup_table_registry::cpi::append_to_lookup_table` with the margin account as authority using its signer seeds.

Instruction: `collect_liquidation_fee`

This instruction collects accrued liquidation fees from a liquidation state, transferring the fee tokens from the margin account to the liquidator. This instruction handles the final settlement of liquidation fees while ensuring the margin account remains healthy after fee collection.

Input parameters

This instruction takes no additional parameters beyond the accounts.

Accounts

- **`margin_account`**: The account that was under liquidation.
  - Signer: No.
  - Init: No.
  - PDA: Yes.
  - Mutable: Yes.
  - Constraints: Must pass health check after fee collection.
- **`liquidator`**: The liquidator account collecting the fees.
  - Signer: Yes.
  - Init: No.
  - PDA: No.
  - Mutable: Yes.
  - Constraints: Must match the liquidator in the liquidation state.
- **`liquidation`**: Account storing the liquidation state with accrued fees.
  - Signer: No.
  - Init: No.
  - PDA: Yes (seeds: `["liquidation", margin_account, liquidator]`).
  - Mutable: Yes.
  - Constraints: No.
- **`liquidator_fee_token`**: The liquidator's token account to receive fees.
  - Signer: No.
  - Init: No.

  - PDA: No.
  - Mutable: Yes.
  - Constraints: Must be owned by the liquidator and match the fee mint and token program.
- **`margin_account_fee_source`**: The margin account's token account to transfer fees from.
  - Signer: No.
  - Init: No.
  - PDA: No.
  - Mutable: Yes.
  - Constraints: Must be owned by the margin account and match the fee mint and token program.
- **`liquidation_fee_mint`**: The mint of the token being used for fee payment.
  - Signer: No.
  - Init: No.
  - PDA: No.
  - Mutable: No.
  - Constraints: Must match the token configuration.
- **`liquidator_fee_token_program`**: The token program for fee transfers.
  - Signer: No.
  - Init: No.
  - PDA: No.
  - Mutable: No.
  - Constraints: Must be a token program.
- **`token_config`**: Configuration for the fee token.
  - Signer: No.
  - Init: No.
  - PDA: Yes.
  - Mutable: No.
  - Constraints: Must be in the same airspace and match the fee mint.
- **`price_oracle`**: Oracle account for fee token pricing.
  - Signer: No.
  - Init: No.
  - PDA: No.
  - Mutable: No.
  - Constraints: Must be a valid Pyth Pull Receiver program account.
- **`redemption_quote_oracle`**: Optional oracle for quote token when using redemption rates.

- Signer: No.
- Init: No.
- PDA: No.
- Mutable: No.
- Constraints: Must be a valid Pyth Pull Receiver program account, if provided.

Additional checks and behavior

- Sets liquidation state to fee-collection mode (`is_collecting_fees = 1`).
- Validates that the fee mint exists in the accrued liquidation fees.
- Requires token configuration to have margin admin type with a valid oracle.
- Constructs price information from oracle accounts for fee valuation.
- Calculates liquidation-fee value using the current token price.
- Applies equity-loss offset logic:
  - If no equity loss, the liquidator receives the full fee.
  - If equity loss exceeds the fee value, the fee is fully absorbed.
  - Otherwise, the fee is reduced by the equity-loss amount.
- Transfers calculated fee tokens from margin account to liquidator.
- Updates margin-account position balance after fee transfer.
- Clears the fee slot in liquidation state.
- Verifies margin-account health after fee collection.

CPI

- Calls `token_interface::transfer_checked` to transfer fee tokens from margin account to liquidator using margin account's signer seeds.

Instruction: `migrate_token_config`

This instruction migrates existing token-configuration accounts from an older version to the current version format. This instruction handles schema migrations by deserializing old configurations, converting them to the new format and reallocating account storage as needed.

Input parameters

This instruction takes no additional parameters beyond the accounts.

Accounts

- **`authority`**: The authority allowed to make configuration changes.
    - Signer: Yes.
    - Init: No.
    - PDA: No.
    - Mutable: No.
    - Constraints: Must be the airspace authority.
- **`airspace`**: The airspace of the config.
    - Signer: No.
    - Init: No.
    - PDA: Yes.
    - Mutable: No.
    - Constraints: Authority must match the airspace authority.
- **`payer`**: The payer for any additional rent costs.
    - Signer: Yes.
    - Init: No.
    - PDA: No.
    - Mutable: Yes.
    - Constraints: Must have sufficient funds to cover rent for reallocation.
- **`mint`**: The mint for the token being migrated.
    - Signer: No.
    - Init: No.
    - PDA: No.
    - Mutable: No.
    - Constraints: No.
- **`token_config`**: The config account to be migrated.
    - Signer: No.
    - Init: No.
    - PDA: Yes (seeds: `["token_config", airspace, mint]`).
    - Mutable: Yes.
    - Constraints: Must be owned by the margin program.
- **`system_program`**: The system program for account operations.
    - Signer: No.
    - Init: No.
    - PDA: No.
    - Mutable: No.
    - Constraints: Must be the system program.

Additional checks and behavior

- Verifies the token-config account is owned by the margin program.
- Deserializes the existing configuration using the old `TokenConfig` format.
- Creates a new configuration struct with
  - all existing fields preserved,
  - the new `token_features` field set to default (empty),
  - the updated version number to `TOKEN_CONFIG_VERSION`, and
  - reserved bytes for future expansion.
- Calculates required rent for the new account size.
- Transfers funds from payer to the config account if additional rent is needed.
- Reallocates the account to the new size using `account.realloc()`.
- Serializes the new configuration format into the reallocated account.

CPI

- May call `anchor_lang::system_program::transfer` if additional rent is required for account reallocation.

5.5. Program: metadata

Instruction: `create_entry`

This instruction creates a new metadata entry account (as a PDA under the airspace) initialized with minimal size.

Input parameters

- `key_account: Pubkey`: The key used with the airspace to derive the entry PDA.
- `space: u64`: Requested space.

Accounts

- **`payer`**: Pays rent to create the entry.
  - Signer: Yes.
  - Init: No.
  - PDA: No.
  - Mutable: Yes.
  - Constraints: Must have sufficient lamports.

- **`authority`**: Airspace authority authorizing the write.
    - Signer: Yes.
    - Init: No.
    - PDA: No.
    - Mutable: No.
    - Constraints: Must equal `airspace.authority`.
- **`airspace`**: The airspace the entry belongs to.
    - Signer: No.
    - Init: No.
    - PDA: No.
    - Mutable: No.
    - Constraints: `airspace.authority == authority.key()`.
- **`metadata_account`**: The entry PDA to create.
    - Signer: No.
    - Init: Yes.
    - PDA: Yes — seeds = `[airspace.key().as_ref(), key_account.as_ref()]`.
    - Mutable: Yes.
    - Constraints: `space = 8` (minimal), owned by this program, and rent-exempt.
- **`system_program`**: The system program.
    - Signer: No.
    - Init: No.
    - PDA: No.
    - Mutable: No.
    - Constraints: Must be the system program.

Additional checks and behavior

- Authorization is enforced by the `airspace.authority == authority` constraint.
- Despite the `space` parameter, the entry is initialized with minimal size (eight bytes); callers must use `set_entry` to allocate actual payload size.

CPI

- Implicit Anchor init uses the system program `create_account` to create the PDA.

Instruction: `set_entry`

This instruction writes arbitrary bytes into an existing entry at `offset`, reallocating the account upward and topping up rent as needed.

Input parameters

- `key_account: Pubkey`: The key used with the airspace to derive/validate the entry PDA.
- `offset: u64`: Byte offset at which to begin writing.
- `data: Vec<u8>`: Bytes to write.

Accounts

- **`payer`**: Pays additional rent if the entry grows.
  - Signer: Yes.
  - Init: No.
  - PDA: No.
  - Mutable: Yes.
  - Constraints: Must have sufficient lamports to cover any rent top-up.
- **`authority`**: Airspace authority authorizing the mutation.
  - Signer: Yes.
  - Init: No.
  - PDA: No.
  - Mutable: No.
  - Constraints: Must equal `airspace.authority`.
- **`airspace`**: The airspace the entry belongs to.
  - Signer: No.
  - Init: No.
  - PDA: No.
  - Mutable: No.
  - Constraints: `airspace.authority == authority.key()`.
- **`metadata_account`**: The entry PDA to modify.
  - Signer: No.
  - Init: No.
  - PDA: Yes — seeds = `[airspace.key().as_ref(), key_account.as_ref()]`.
  - Mutable: Yes.
  - Constraints: Must be owned by this program — rent-exempt after any realloc.
- **`system_program`**: The system program.

- Signer: No.
- Init: No.
- PDA: No.
- Mutable: No.
- Constraints: Must be the system program.

Additional checks and behavior

- Computes `new_len = offset + data.len()` — on integer overflow, the instruction errors with `ErrorCode::ConstraintSpace`.
- If `metadata_account.data_len() < new_len`,
  - computes required lamports using `Rent::minimum_balance(new_len)`,
  - transfers any shortfall from `payer` to `metadata_account`, and
  - calls `metadata_account.realloc(new_len, /*zero_init=*/false)`. Newly added bytes are *not* zeroed.
- Performs an in-place copy: `metadata[offset .. offset+data.len()] = data`.
- Shrinking is not performed; if `new_len` ≤ current length, the account size remains unchanged and only the slice is overwritten.
- Content is opaque; no schema validation is performed by this program.

CPI

- System program `transfer` (to top up rent when growing).
- Uses Anchor's in-program `realloc` on the PDA (no external CPI for realloc).

Instruction: `remove_entry`

This instruction deletes an entry by manually closing its PDA and transferring all lamports to a receiver.

Input parameters

- `key_account: Pubkey`: The key used with the airspace to derive/validate the entry PDA.

Accounts

- **`receiver`**: Destination for reclaimed lamports.
  - Signer: No.
  - Init: No.

    - PDA: No.
    - Mutable: Yes.
    - Constraints: Expected to be a wallet; only receives lamports.
- **`authority`**: Airspace authority authorizing the deletion.
    - Signer: Yes.
    - Init: No.
    - PDA: No.
    - Mutable: No.
    - Constraints: Must equal `airspace.authority`.
- **`airspace`**: The airspace the entry belongs to.
    - Signer: No.
    - Init: No.
    - PDA: No.
    - Mutable: No.
    - Constraints: `airspace.authority == authority.key()`.
- **`metadata_account`**: The entry PDA to remove.
    - Signer: No.
    - Init: No.
    - PDA: Yes — seeds = `[airspace.key().as_ref(), key_account.as_ref()]`.
    - Mutable: Yes.
    - Constraints: Must be owned by this program.

Additional checks and behavior

- Manually closes the PDA. Drains lamports from `metadata_account` to `receiver` — `assign(&system_program::ID)` to clear ownership and `realloc(0, /*zero_init=*/false)` to zero-length data.
- No event is emitted; content is considered opaque and is not cleared beyond ownership reset and size 0.

CPI

None.

6. Assessment Results

During our assessment on the scoped Glow Protocol programs, we discovered 10 findings. No critical issues were found. Five findings were of high impact, four were of medium impact, and one was of low impact.

Based on the number of severe findings uncovered during the audit, it is our opinion that the project is not yet ready for production. We strongly advise a comprehensive reassessment before deployment to help identify any potential issues or vulnerabilities introduced by necessary fixes or changes.

We strongly recommend increasing test coverage to include both positive and negative test cases for all instructions and business logic. Positive cases should validate that your code behaves correctly under normal conditions and produces expected outputs, while negative cases should verify that your system gracefully handles edge cases, invalid inputs, and error conditions without failing unexpectedly.

6.1. 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.