Etherfuse Stablebond

Security Assessment

August 29th, 2024 — Prepared by OtterSec

Ajay Shankar Kunapareddy d1r3wolf@osec.io

Akash Gurugunti sud0u53r.ak@osec.io

Robert Chen r@osec.io

Table of Contents

Appendices

01 — Executive Summary

Overview

Etherfuse engaged OtterSec to assess the `stablebond` program. This assessment was conducted between August 17th and August 24th, 2024. For more information on our auditing methodology, refer to Appendix B.

Key Findings

We produced 13 findings throughout this audit engagement.

In particular, we identified several high-risk vulnerabilities, including the lack of discriminator verification, allowing the possibility of an attacker substituting the intended account with another account of a similar structure (OS-SVB-ADV-01), and the possibility of bypassing the lamports-based check for token account initialization by tricking the program into thinking that a specific account already holds lamports, thus skipping the initialization process (OS-SVB-ADV-02). We also highlighted the accumulation of the interest rate even when no issuance is active, resulting in artificially inflated bond prices and enabling users to receive extra rewards for periods without an active issuance (OS-SVB-ADV-04).

Furthermore, while initializing and starting issuances, there is no check for existing active issuances, allowing multiple overlapping issuances on the same bond, which may result in improper reward distribution (OS-SVB-ADV-00). Additionally, the issuance payout process fails to validate the current timestamp and whether an issuance is associated with the correct bond, enabling premature closure of the issuance and incorrect bond issuances, respectively (OS-SVB-ADV-05).

We also made recommendations to ensure adherence to coding best practices (OS-SVB-SUG-00) and suggested the removal of unutilized and redundant code within the system for increased readability (OS-SVB-SUG-03, OS-SVB-SUG-04). We further advised incorporating additional checks within the codebase for improved robustness and security (OS-SVB-SUG-02).

02 — Scope

The source code was delivered to us in a Git repository at https://github.com/etherfuse/stablebond. This audit was performed against commit 40f8eb3.

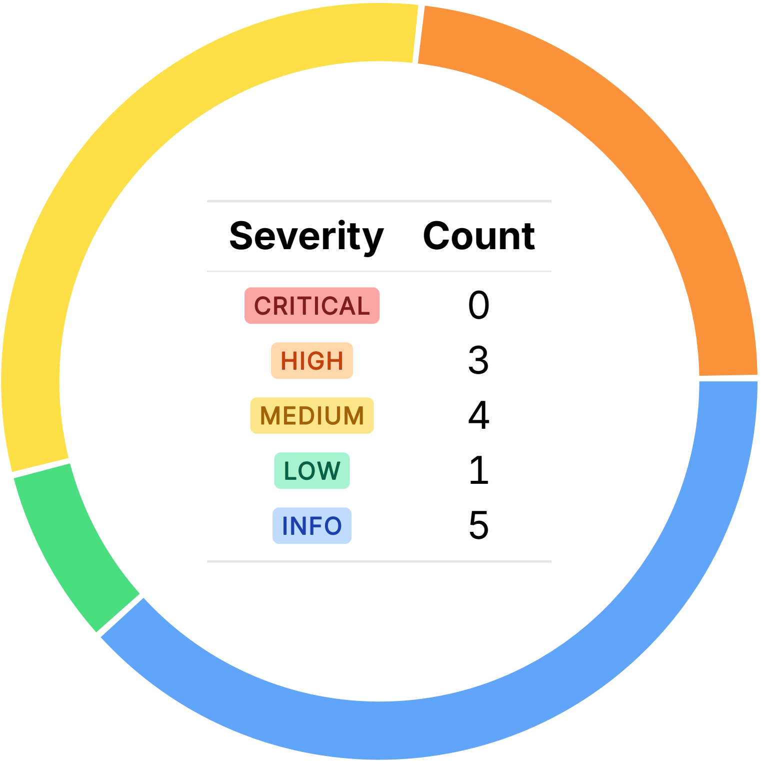
A brief description of the program is as follows:

Name	Description
stablebond	A yield-bearing stablebond which utilizes the token 2022 extension, allowing it to accrue interest while it is held.

03 — Findings

Overall, we reported 13 findings.

We split the findings into **vulnerabilities** and **general findings**. Vulnerabilities have an immediate impact and should be remediated as soon as possible. General findings do not have an immediate impact but will aid in mitigating future vulnerabilities.



Severity	Count
CRITICAL	0
HIGH	3
MEDIUM	4
LOW	1
INFO	5

04 — Vulnerabilities

Here, we present a technical analysis of the vulnerabilities we identified during our audit. These vulnerabilities have *immediate* security implications, and we recommend remediation as soon as possible.

Rating criteria can be found in Appendix A.

ID	Severity	Status	Description
OS-SVB-ADV-00	HIGH	RESOLVED ✓	The `InitializeIssuance` and `StartIssuance` instructions do not check for existing active issuances, allowing multiple overlapping issuances on the same bond, which may result in improper reward distribution.
OS-SVB-ADV-01	HIGH	RESOLVED ✓	`unpack_from_slice` lacks discriminator verification, allowing the possibility of an attacker substituting the intended account with another account of a similar structure.
OS-SVB-ADV-02	HIGH	RESOLVED ✓	The lamports-based check for token account initialization may be bypassed by tricking the program into thinking that a specific account already holds lamports, thus skipping the initialization process.
OS-SVB-ADV-03	MEDIUM	RESOLVED ✓	`process_redeem_purchase_order` does not verify the issuance account's index via seeds, allowing an attacker to provide an outdated issuance and purchase bonds at a lower price. Also, `user_payment_token_account_info` is missing any account checks `process_create_purchase_order`.

OS-SVB-ADV-04	MEDIUM	RESOLVED ✓	The protocol incorrectly allows the interest rate to accumulate even when no issuance is active, resulting in artificially inflated bond prices and enabling users to receive extra rewards for periods without an active issuance.
OS-SVB-ADV-05	MEDIUM	RESOLVED ✓	`process_payout_issuance` fails to validate the current timestamp and whether an issuance is associated with the correct bond, enabling premature closure of the issuance and incorrect bond issuances, respectively.
OS-SVB-ADV-06	MEDIUM	RESOLVED ✓	`process_initialize_bond` allows bond tokens to be initialized with decimals other than six, while `mint_to_instruction` and `get_bond_price` expect the decimals to be exactly six.
OS-SVB-ADV-07	LOW	RESOLVED ✓	The `MAX_AGE_OF_PRICE_FEED` constant is set to one hour, which may be too long and may expose the system to outdated or stale price feeds.

Overlapping of Multiple Issuances Simultaneously HIGH OS-SVB-ADV-00

Description

The `InitializeIssuance` and `StartIssuance` instructions do not check if there is an active issuance already running under the bond before initiating a new one. This oversight allows multiple issuances to be active simultaneously. The design flaw allows the possibility of multiple overlapping issuances under the same bond. Since all issuances are tied to the same bond token, each issuance accrues interest over time. If multiple issuances overlap, it complicates the distribution of interest and rewards because the interest calculation and reward distribution mechanisms are designed to handle a single active issuance at a time.

>_ *src/commands/initialize_issuance.rs* RUST

```rust
pub fn handle_initialize_issuance(args: InitializeIssuanceArgs) -> Result<()> {
    [...]
    let estimated_start_datetime = if bond.issuance_number == 0 {
        match args.estimated_start_datetime {
            Some(estimated_start_datetime) => estimated_start_datetime,
            None => bail!("Estimated start datetime required for first issuance"),
        }
    } else {
        match args.estimated_start_datetime {
            Some(estimated_start_datetime) => bail!(
                "Estimated start datetime only allowed for first issuance, provided {}",
                estimated_start_datetime
            ),
            None => {
                let issuance_account = find_issuance_pda(bond_account, bond.issuance_number).0;
                let data = config.client.get_account_data(&issuance_account)?;
                let issuance = Issuance::from_bytes(&data).unwrap();
                issuance.actual_start_datetime + issuance.length_in_seconds
            }
        }
    };
    [...]
}
```

With overlapping issuances, users who hold bond tokens from different issuances may receive improper or incorrect distributions of rewards. For example, interest accrued for a bond might be incorrectly allocated across multiple issuances, resulting in improper distribution among bondholders.

Furthermore, the `CollectPayment` instruction allows delegators to collect payments from a bond issuance. However, the current implementation restricts the delegator to collecting payments only from the latest active issuance. This restriction may result in a scenario where user funds are locked if a new issuance starts before the previous one is fully settled and users have purchased bonds in both issuances.

Remediation

Include a check in `InitializeIssuance` to ensure no active `issuance` is currently running. Specifically, for any new `issuance` (`issuance_number > 0`), before initializing a new `issuance`, calculate the `end_time` of the latest active issuance. Ensure that the `estimated_start_datetime` of the new `issuance` is after this `end_time`. Also, modify the `CollectPayment` instruction to allow payment collection from any active issuance in which the user holds bonds, not just the latest.

Patch

Resolved in #104.

Discriminator Check Bypass HIGH OS-SVB-ADV-01

Description

The vulnerability concerns the lack of discriminator verification in all instances of `unpack_from_slice` within the codebase. Discriminators are utilized to differentiate between different account types. `unpack_from_slice` performs deserialization of the account data but does not explicitly check the discriminator. Thus, instructions may be exploited by passing one account in place of another.

>_ *src/state/payment.rs* RUST

```rust
fn unpack_from_slice(src: &[u8]) -> Result<Self, ProgramError> {
    let mut p = src;
    let payment = Self::deserialize(&mut p).map_err(|_| {
        msg!("Failed to deserialize Payment");
        ProgramError::InvalidAccountData
    })?;
    if !payment.is_initialized() {
        return Err(ErrorCode::PaymentNotInitialized.into());
    }
    Ok(payment)
}
```

As an example, in `process_purchase_bond`, it is possible to substitute a `Payout` account for a `Payment` account. Since both `Payout` and `Payment` have the issuance key in the same position, the issuance check will be bypassed as no discriminator checks are performed to differentiate between these two account types. Consequently, the payment tokens sent by the user will be transferred to the `Payout` account instead. Although the attacker wouldn't be able to directly collect these tokens if the Payout account is not set up to do so, they would still successfully obtain the bonds.

Remediation

Ensure that the `discriminator` is checked in `unpack_from_slice` to verify the account type before processing it. This helps to ensure that the account data is of the expected type.

Patch

Resolved in ed8cf39.

Bypassing Token Account Initialization HIGH OS-SVB-ADV-02

Description

In the current implementation of `bond::process_purchase_bond` and `purchase_order::process_redeem_purchase_order` (shown below), there may be a potential vulnerability in method of determining the initialization status of a token account (`user_token_account_info`). These functions check if the lamports balance of `user_token_account_info` is zero to determine if the token account is initialized. If the balance is zero, the account is assumed to be uninitialized, and the program proceeds to create the token account.

>_ *src/processor/purchase_order.rs* RUST

```rust
pub fn process_redeem_purchase_order(
    _program_id: &Pubkey,
    accounts: &[AccountInfo],
) -> ProgramResult {
    [...]
    // create token account if needed for the bonds
    {
        if user_token_account_info.lamports() == 0 {
            invoke(
                &Bond::create_token_account_instruction(
                    user_wallet_account_info.key,
                    mint_account_info.key,
                    token_2022_program_info.key,
                ),
                &[
                    [...]
                ],
            )?;
        }
    }
    [...]
}
```

However, it is possible to exploit this particular method by sending a small amount of lamports to `user_token_account_info` before the user attempts to execute `process_purchase_bond` or `process_redeem_purchase_order`. Since the account will then have a non-zero lamport balance, the program will incorrectly assume that the account is already initialized and skip the initialization process.

Remediation

Check the `data_len` of `user_token_account_info` instead of checking the lamports balance. The `data_len` property reflects the size of the data stored in the account, which will be non-zero if the account is initialized properly with a token program.

Patch

Resolved in cf2fc93.

Absence of Verification of Issuance Account Index MEDIUM OS-SVB-ADV-03

Description

In `purchase_order::process_redeem_purchase_order`, it is not verified that the provided issuance account corresponds to a valid and current issuance. Specifically, it does not check if the index of the issuance account matches the expected value derived from deterministic seeds. An attacker may provide an old issuance account, where the bond price was lower due to previous market conditions or earlier issuance terms. Since the issuance index is not validated against the current issuance number stored in `bond`, the system will incorrectly accept this outdated issuance as valid. Thus, by purchasing the bonds at a lower price, the attacker would make an instant profit.

>_ *src/processor/purchase_order.rs* RUST

```rust
pub fn process_redeem_purchase_order(
    _program_id: &Pubkey,
    accounts: &[AccountInfo],
) -> ProgramResult {
    [...]
    // #[account(3, name = "issuance_account")]
    validate!(
        issuance_account_info.owner == &crate::ID,
        ErrorCode::InvalidIssuanceOwner
    )?;
    let issuance = Issuance::unpack(&issuance_account_info.data.borrow())?;
    validate!(
        issuance.status == IssuanceStatus::Started || issuance.status ==
            ↪ IssuanceStatus::Matured,
        ErrorCode::InvalidIssuanceStatus
    )?;
    validate!(
        issuance.parent_bond == *bond_account_info.key,
        ErrorCode::InvalidIssuance
    )?;

    Ok(())
}
```

Additionally, `process_create_purchase_order` is missing account checks for the `user_payment_token_account_info` account. As a result of this, it is possible for `user_payment_token_account_info` to be set to another user's account.

Remediation

Verify the issuance index with `bond.issuance_number` and add account checks for `user_payment_token_account_info`.

Patch

Resolved in 6850535.

Interest Rate Accumulation MEDIUM OS-SVB-ADV-04

Description

The issue revolves around the improper handling of interest rate accumulation, specifically during periods when no active issuance is taking place. The protocol allows buying bonds during one issuance and redeeming them in another. However, the rate for the interest-bearing extension does not reset to zero after an issuance is completed, resulting in the interest-bearing rate continuing to accumulate. This creates a scenario where, even when there is no active issuance, the rate increases, artificially inflating bond prices. Thus, users receive extra rewards for the time when no issuance is active.

Remediation

Ensure that all interest-bearing extensions are properly reset to zero after each issuance ends.

Patch

Resolved in cf2fc93.

Lack of Timestamp and Bond Validation MEDIUM OS-SVB-ADV-05

Description

`issuance::process_payout_issuance` fails to check whether the current timestamp has passed the expected maturity time of the issuance before changing its status to *Matured*. This oversight may result in several problems. Normally, an issuance has a defined maturity period. This period is determined by the `actual_start_datetime` and `length_in_seconds` fields, which together specify the exact time when the issuance should mature. In `process_payout_issuance`, the current timestamp is not checked to be greater than `actual_start_datetime + length_in_seconds`, and as a result, the function may prematurely close an issuance, even if the intended maturity date has not been reached.

>_ *src/processor/issuance.rs* RUST

```rust
pub fn process_payout_issuance(_program_id: &Pubkey, accounts: &[AccountInfo]) -> ProgramResult
    ↪ {
    [...]
    // #[account(0, name = "config_account")]
    validate!(
        config_account_info.key == &Config::address(),
        ErrorCode::InvalidConfigAddress
    )?;
    validate!(
        *config_account_info.owner == crate::ID,
        ErrorCode::InvalidConfigOwner
    )?;
    // #[account(1, name = "delegate_account")]
    validate!(
        delegate_account_info.key == &Delegate::address(delegate_wallet_info.key),
        ErrorCode::InvalidDelegateAddress
    )?;
    validate!(
        delegate_account_info.data_len() != 0,
        ErrorCode::DelegateNotInitialized
    )?;
    validate!(
        *delegate_account_info.owner == crate::ID,
        ErrorCode::InvalidDelegateOwner
    )?;
    [...]
}
```

As this instruction changes the issuance status to *Matured*, users who hold bond tokens associated with this issuance and expect to redeem them after the issuance has matured will be unable to redeem their bond tokens for that issuance. Thus, due to the function's lack of timestamp validation, an issuance may be marked as mature prematurely, disrupting the proper flow of payouts and impacting the overall integrity of the issuance system.

Furthermore, `process_payout_issuance` does not validate whether the issuance is associated with the correct bond by checking `issuance.parent_bond`. This missing check implies that the function may process payouts for an issuance that does not belong to the specified bond account (`bond_account_info`).

Remediation

Add a check to ensure that the current timestamp is greater than the calculated maturity timestamp (`actual_start_datetime + length_in_seconds`) before proceeding with the status change. Additionally, include a check to ensure that `issuance.parent_bond == *bond_account_info.key` in `process_payout_issuance`.

Patch

Resolved in #104.

Mismatch In Expected Decimal Value Resulting In DOS MEDIUM OS-SVB-ADV-06

Description

`bond::process_initialize_bond` is responsible for initializing a bond token, including setting up the token's decimals. The decimals may be any value between 0 and 19, as provided by the input during the initialization process. However, in `PaymentFeed::get_bond_price` and `Bond::mint_to_instruction` , the bond's decimals are expected to be exactly six. This is enforced through validation checks.

>_ *src/processor/bond.rs* RUST

```rust
#[allow(unused_variables)]
pub fn process_initialize_bond(
    _program_id: &Pubkey,
    accounts: &[AccountInfo],
    data: InitializeBondInstruction,
) -> ProgramResult {
    [...]
    let InitializeBondInstruction {
        decimals,
        [...]
    } = data;

    validate!(
        decimals <= 19,
        ErrorCode::InvalidArgument,
        "Decimals must be less than or equal to 19"
    )?;
    [...]
}
```

Consequently, if a bond is initialized with a decimal value different from six within `process_initialize_bond`, subsequent calls to `get_bond_price` and `mint_to_instruction` will fail due to these strict validation checks. This will result in a Denial of Service (DoS) scenario as the bond initialization will fail.

A similar issue exists in `payment_feed::get_bond_price`. `process_start_issuance` in `issuance` derives the `scaling_factor` from `Bond::DECIMALS`, which is hard-coded to six. In `get_bond_price`, however, when calculating `ui_bond_price` utilizing `amount_to_ui_amount` , the `scaling_factor` is taken from an external source (derived from `payment_decimals`). If `payment_decimals` is not equal to six, the scaling factor will differ from the one utilized in `process_start_issuance`, resulting in the calculation of an incorrect bond price.

Remediation

Ensure consistency in how decimals are handled across the entire application. If the intention is for bonds to strictly have six decimals, this should be enforced during bond initialization in `process_initialize_bond`, by hard-coding the decimals to `Bond::DECIMALS` instead of accepting them as user input.

Patch

Resolved in #104.

Excessively Long Price Feed Expiry Period LOW OS-SVB-ADV-07

Description

In the protocol, the current setting of one hour for `MAX_AGE_OF_PRICE_FEED` implies that price data up to one hour old is still considered valid and usable. A longer maximum age allows for more time to pass before data is considered stale. This may be problematic in volatile markets where prices may change rapidly. Relying on outdated prices reduces the reliability and effectiveness of the protocol. A typical timeframe for maximum safety in price feeds is around 5 minutes.

>_ *src/constants.rs* RUST

```rust
cfg_if::cfg_if! {
    if #[cfg(feature = "mainnet-beta")] {
        pub const MAX_AGE_OF_PRICE_FEED: u64 = 60 * 60; // 1 hour
    } else {
        pub const MAX_AGE_OF_PRICE_FEED: u64 = 60 * 60 * 24 * 7; // 1 week
    }
}
```

Remediation

Ensure the price feed becomes stale after a shorter time period. For comparison, Solend utilizes a much shorter period with their `STALE_AFTER_SLOTS_ELAPSED` constant set to 240 slots, roughly equivalent to 2 minutes.

Patch

Resolved in #104.

05 — General Findings

Here, we present a discussion of general findings during our audit. While these findings do not present an immediate security impact, they represent anti-patterns and may result in security issues in the future.

ID	Description
OS-SVB-SUG-00	Suggestions regarding inconsistencies in the codebase and ensuring adherence to coding best practices.
OS-SVB-SUG-01	There are several instances where proper validation is not performed, resulting in potential security issues.
OS-SVB-SUG-02	Additional safety checks may be incorporated within the codebase to make it more robust and secure.
OS-SVB-SUG-03	The codebase contains multiple cases of redundancy that should be removed for better maintainability and clarity.
OS-SVB-SUG-04	Recommendations to remove unnecessary code from CPI calls.

Code Maturity

OS-SVB-SUG-00

Description

1. There is a lack of functionality to pause or remove a delegate, which may be crucial in scenarios such as a delegate key compromise. Adding this capability will improve the program's security by allowing the system to quickly respond to and mitigate potential threats.
2. In the current implementation, signer seeds are being initialized multiple times instead of re-utilizing them. For example, in the `Config` account, the seeds in `process_initialize_config` are initiated six times, impacting efficiency.
3. Within the codebase, `Discriminator` is spelled incorrectly in the structure name as `Descriminator`.

   >_ *src/generated/types/descriminator.rs* RUST

   ```rust
   #[derive(BorshSerialize, BorshDeserialize, Clone, Debug, Eq, PartialEq, PartialOrd, Hash)]
   #[cfg_attr(feature = "serde", derive(serde::Serialize, serde::Deserialize))]
   pub enum Descriminator {
       [...]
   }
   ```

4. Currently, there is no common functionality for verifying account initializations to eliminate redundant checks such as writable status, `data_is_empty` check, ownership by the system program, signer verification (if not a `PDA`), and address checks for `PDA`s. These checks are implemented multiple times across different parts of the codebase.

Remediation

1. Add functionality for pausing and removing a delegate.
2. Maintain a `signer_seeds` function on data structures to reduce code redundancy and improve maintainability within the codebase.
3. Ensure the structure name is spelled correctly.
4. Implement a common utility function to encapsulate all the account initialization checks to improve code clarity and ensure that no checks are missed during future development.

Missing Validation Logic OS-SVB-SUG-01

Description

1. Utilize `unpack` over `unpack_from_slice` in all cases to ensure that accounts are correctly initialized with the appropriate size and layout. In addition to deserializing data, `unpack` performs a check to ensure that the length of the byte slice matches the expected size of the deserialized structure.
2. Add a check to ensure that `InitializeBondInstructionArgs.payment_feed_type` is not equal to `PaymentFeedType::Stub` during bond initialization. Also, include version and initialization checks for the `AccessPass` structure.
3. To verify that an issuance has not already matured, add a check in `process_purchase_bond` to ensure `issuance.status == Started`.
4. Implement address checks utilizing seeds for `PDA` s, `ATA` s, etc., wherever possible to catch errors earlier, rather than allowing them to be reverted later during `CPI` calls. For example, for `AccessPass` instructions, the access pass address may be checked utilizing the seeds early on.
5. Utilize `payment_feed::check_payment_feed_accounts` to verify the payment feed along with feed accounts, as the current checks seem inefficient in comparison.

Remediation

Incorporate the above validations into the codebase.

Additional Safety Checks

OS-SVB-SUG-02

Description

1. The writable account checks may not be stringent in the codebase currently, relying on `CPI` calls to catch errors if an account is not writable when expected. These checks are performed indirectly through error handling in `CPI` calls, which implies issues may not be immediately obvious or may only surface during more complex interactions.
2. In the current implementation of `purchase_order::process_create_purchase_order`, there is no validation to ensure that the `nft_collection_mint` specified in the `config` matches the `nft_collection_mint_account_info.key`. As a result, any `NFT` collection mint may be utilized.
3. In `PurchaseBond`, when handling payments or transactions, the system does not verify if the payment token accounts are Associated Token Accounts (ATAs). Thus, it does not ensure that the payment accounts utilized are properly set up as ATAs for the specific token involved. Therefore, the admin ends up tracking all token accounts, including those that may not be properly set up or managed, resulting in additional overhead and management complexity for the admin.
4. The program does not ensure the reliability of the prices fetched from Pyth and Switchboard due to a lack of threshold and price confidence checks. This may result in the application utilizing inaccurate or unreliable price data, negatively affecting the overall system integrity. Additionally, the account owner check is missing in `SwitchboardV2PriceFeed::load_checked`.

>_ *src/state/oracle.rs* RUST

```rust
pub fn load_checked(
    ai: &AccountInfo,
    current_time: i64,
    max_age: u64,
) -> Result<Self, ErrorCode> {
    let price_feed = load_pyth_price_feed(ai)?;
    let ema_price = price_feed
        .get_ema_price_no_older_than(current_time, max_age)
        .ok_or(ErrorCode::StaleOracle)?;
    [...]
}
```

Remediation

1. Raise custom errors when writable checks fail to provide immediate feedback. This allows faster identification of issues with account permissions or state, without waiting for errors to propagate through CPI calls.
2. Check that `config.nft_collection_mint` is equal to `nft_collection_mint_account_info.key` in `purchase_order::process_create_purchase_order`.
3. Implement checks in the `PurchaseBond` process to verify that the payment token accounts are ATAs.
4. Ensure that the price data meets a minimum threshold of confidence to help avoid issues arising from incorrect or volatile price feeds. Also, add the account owner check in `SwitchboardV2PriceFeed::load_checked`.

Unutilized Code OS-SVB-SUG-03

Description

The following functions, variables, or accounts are either redundant or not utilized and may be removed:

1. The `token_metadata` variable in `process_initialize_bond` is redundant.
2. In `process_start_issuance`, `ending_token_amount` is updated twice on issuance.
3. The `config_account_info` account is unutilized in `process_payout_issuance` and `process_purchase_bond`.
4. In `Issuance::is_currently_within_window` and `Bond::mint_to_instruction`, the `length_in_seconds` argument and the `mint` argument, respectively, are redundant as they may be acquired from `self` instead.
5. Multiplying the `scaling_factor` by one in `PaymentFeedType::get_bond_price` is redundant.
6. In `process_add_access_pass`, the check to verify that `user_token_account.owner` is equal to `*user_wallet_account_info.key` is performed twice.
7. In `process_initialize_bond`, the `token_metadata` is defined but not utilized.

Remediation

Remove all above-mentioned cases of unutilized or redundant code.

Redundant Code in CPI Calls OS-SVB-SUG-04

Description

1. In `process_request_redemption` (shown below) and `process_create_purchase_order`, the `NftIssuanceVault::create_account_instruction` and `PurchaseOrder::create_account_instruction` `CPI` calls, respectively, do not require `nft_mint_account_info` and `token_program_info` accounts.
2. In `process_request_redemption` (shown below), it is not necessary to pass `config_account_info` to `NftIssuanceVault::create_nft_mint` `CPI` call. Similarly, `user_account_info` is unnecessary for the `NftIssuanceVault::mint_nft` `CPI` call.

>_ *src/processor/bond.rs* RUST

```rust
pub fn process_request_redemption(
    _program_id: &Pubkey,
    accounts: &[AccountInfo],
    data: RequestRedemptionInstruction,
) -> ProgramResult {
    [...]
    // Create the Nft Issuance Vault Token account for the bonds
    {
        let seed = NftIssuanceVault::seed();
        let bump = NftIssuanceVault::address_with_bump(nft_mint_account_info.key).1;
        let signer_seeds = &[seed, nft_mint_account_info.key.as_ref(), &[bump]];
        invoke_signed(
            &NftIssuanceVault::create_nft_token_account_instruction(
                user_account_info.key,
                nft_mint_account_info.key,
                mint_account_info.key,
            ),
            &[
                user_account_info.clone(),
                nft_issuance_vault_token_account_info.clone(),
                nft_issuance_vault_account_info.clone(),
                mint_account_info.clone(),
            ],
            &[signer_seeds],
        )?;
    }
    [...]
}
```

3. There are multiple instances where `invoke_signed` is unnecessarily utilized instead of `invoke` , adding more complexity. For the following cross program invocations, `invoke` maybe utilized rather than `invoke_signed`:
    (a) `NftIssuanceVault::create_nft_mint` and `NftIssuanceVault::create_nft_token_account_instruction` `CPI` calls in `process_request_redemption`.
    (b) `Config::create_mint` and `Config::create_token_account_instruction` `CPI` calls in `process_initialize_config`.
    (c) `Payment::create_token_account_instruction` and `Payout::create_token_account_instruction` `CPI` calls in `process_initialize_issuance`.
    (d) `PurchaseOrder::create_nft_mint` `CPI` call in `process_create_purchase_order`.
4. It is not necessary to add the authority to the `signer_pubkeys` array utilized for `MultiSig` in the below-listed `CPI` calls:
    (a) In `process_purchase_bond` - `transfer_checked` `CPI` call.
    (b) In `process_redeem_bond` - `transfer_checked` `CPI` call.
    (c) In `process_collect_payment` - `transfer_checked` `CPI` call.
    (d) In `process_start_issuance` - `update_rate` `CPI` call.
    (e) In `process_create_purchase_order` - `transfer_checked` `CPI` call.

Remediation

Ensure the unutilized or redundant code is removed from the above-stated instances.

A — Vulnerability Rating Scale

We rated our findings according to the following scale. Vulnerabilities have immediate security implications. Informational findings may be found in the General Findings.

CRITICAL

Vulnerabilities that immediately result in a loss of user funds with minimal preconditions.

Examples:

- Misconfigured authority or access control validation.
- Improperly designed economic incentives leading to loss of funds.

HIGH

Vulnerabilities that may result in a loss of user funds but are potentially difficult to exploit.

Examples:

- Loss of funds requiring specific victim interactions.
- Exploitation involving high capital requirement with respect to payout.

MEDIUM

Vulnerabilities that may result in denial of service scenarios or degraded usability.

Examples:

- Computational limit exhaustion through malicious input.
- Forced exceptions in the normal user flow.

LOW

Low probability vulnerabilities, which are still exploitable but require extenuating circumstances or undue risk.

Examples:

- Oracle manipulation with large capital requirements and multiple transactions.

INFO

Best practices to mitigate future security risks. These are classified as general findings.

Examples:

- Explicit assertion of critical internal invariants.
- Improved input validation.

B — Procedure

As part of our standard auditing procedure, we split our analysis into two main sections: design and implementation.

When auditing the design of a program, we aim to ensure that the overall economic architecture is sound in the context of an on-chain program. In other words, there is no way to steal funds or deny service, ignoring any chain-specific quirks. This usually requires a deep understanding of the program's internal interactions, potential game theory implications, and general on-chain execution primitives.

One example of a design vulnerability would be an on-chain oracle that could be manipulated by flash loans or large deposits. Such a design would generally be unsound regardless of which chain the oracle is deployed on.

On the other hand, auditing the program's implementation requires a deep understanding of the chain's execution model. While this varies from chain to chain, some common implementation vulnerabilities include reentrancy, account ownership issues, arithmetic overflows, and rounding bugs.

As a general rule of thumb, implementation vulnerabilities tend to be more "checklist" style. In contrast, design vulnerabilities require a strong understanding of the underlying system and the various interactions: both with the user and cross-program.

As we approach any new target, we strive to comprehensively understand the program first. In our audits, we always approach targets with a team of auditors. This allows us to share thoughts and collaborate, picking up on details that the others may have missed.

While sometimes the line between design and implementation can be blurry, we hope this gives some insight into our auditing procedure and thought process.