**Jet Governance**

# Audit

Presented by:

**OtterSec** contact@osec.io

**David Chen** ra@osec.io

**Robert Chen** notdeghost@osec.io

# Table of Contents

# 01 | Executive Summary

## Overview

Jet Protocol engaged OtterSec to perform an assessment of the `jet-governance` program prior to deployment.

This assessment was conducted between February 23rd and March 6th, 2022, with a focus on the following:

- Ensure integrity of the Jet Governance program at an implementation and design level
- Analyze the program attack surface and provide meaningful recommendations to mitigate future vulnerabilities

Critical vulnerabilities were communicated to the team prior to the delivery of the report to speed up remediation. After delivering our audit report, we worked closely with the Jet Protocol team over the course of two weeks to streamline patches and confirm remediation.

While some issues such as the lack of a finalization check (OS-JET-ADV-03) were relatively easy to patch, others such as OS-JET-ADV-00 and OS-JET-ADV-02 required an in-depth discussion to properly rework the economic design.

We delivered final confirmation of the patches March 25th, 2022.

## Key Findings

The following is a summary of the major findings in this audit.

- 13 findings total
- 4 vulnerabilities which could lead to loss of funds
    - OS-JET-ADV-00: Unsafe conversion design
    - OS-JET-ADV-01: Improper conversion rounding
    - OS-JET-ADV-02: Unsafe unbond cache design
    - OS-JET-ADV-03: Missing airdrop finalization check

As part of this audit, we also provided proof of concepts for each vulnerability to prove exploitability and enable simple regression testing. These scripts can be found at https://osec.io/pocs/jet-governance. For a full list, see Appendix C.

We also observed the following:

- Code quality of the program was high and overall design was solid
- The team was very knowledgeable and responsive to our feedback
- The use of Anchor and a comprehensive internal audit mitigated many implementation level bugs

## 02 | Scope

We received the program from Jet Protocol and began the audit February 23, 2022. The source code was delivered to us in a git repository at https://github.com/jet-lab/jet-governance/. This audit was performed against commit 6b7139e.

There were a total of three programs included in this audit. A brief description of the three programs is as follows. A full list of program files and hashes can be found in Appendix A.

| Name | Description |
|---|---|
| Auth | Responsible for maintaining know-your-customer requirements for users |
| Rewards | Distribution of rewards to users. There are three different types of rewards: airdrop, award, and distribution. |
| Staking | Allows users to stake tokens in exchange for vote tokens which are then used in the governance program. |

As part of Jet's deployment sequence, we also received an internal audit report prepared by three engineers not involved in the implementation of the program. While the findings from the internal report were not yet integrated into the codebase, we treated them as known issues and operated as if a patch had already been deployed.

# 03 | 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 onchain program. In other words, there is no way to steal tokens or deny service, ignoring any Solana specific quirks such as account ownership issues. An example of a design vulnerability would be an onchain oracle which could be manipulated by flash loans or large deposits.

On the other hand, auditing the implementation of the program requires a deep understanding of Solana's execution model. Some common implementation vulnerabilities include account ownership issues, arithmetic overflows, and rounding bugs. For a non-exhaustive list of security issues we check for, see Appendix B. One such issue that was identified here allowed a malicious user to drain the entire pool of staked tokens due to a rounding bug (OS-JET-ADV-01).

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. For example, allowing a user to easily manipulate conversion rates between tokens and shares was shown to be economically unsound (OS-JET-ADV-00).

As we approach any new target, we strive to get a comprehensive understanding of the program first. In our audits, we always approach any target in a team of two. This allows us to share thoughts and collaborate, picking up on details that the other missed.

We also drew ideas from the internal audit we were provided. At the same time, we didn't take the results for-granted, independently verifying them. We discovered one discrepancy with the internal audit which would allow an attacker to underflow a reward calculation, although this wasn't exploitable due to additional assertions (OS-JET-SUG-05).

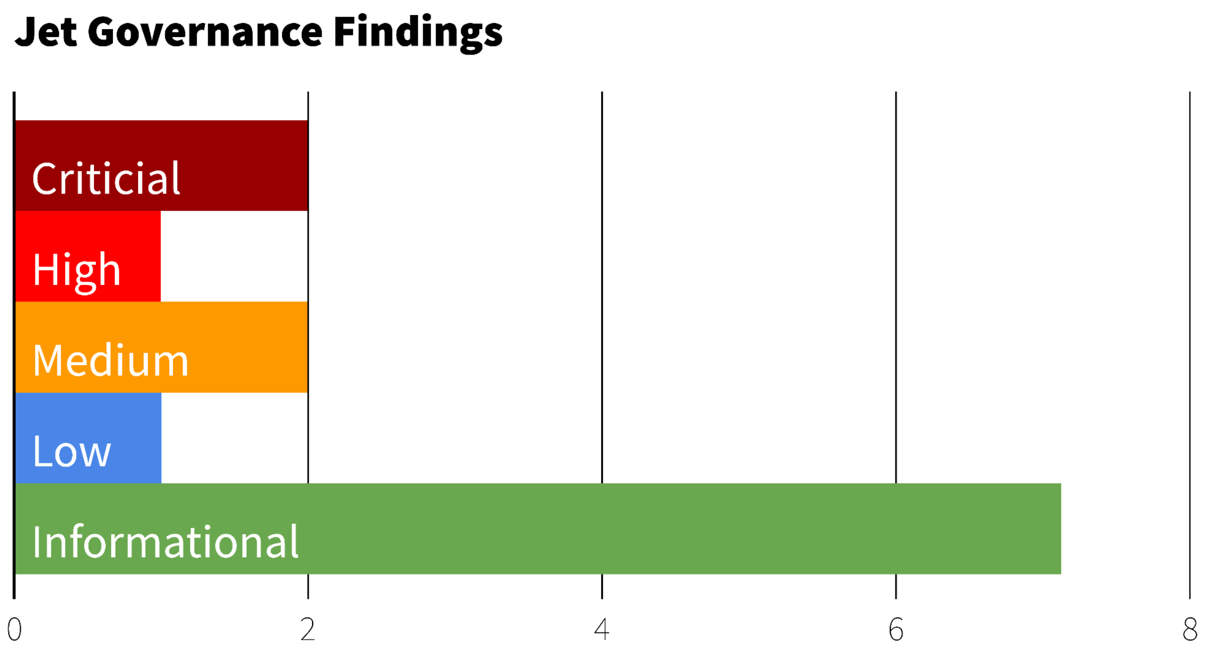
While sometimes the line between design and implementation can be blurry, we hope this gives some insight into our auditing procedure and thought process.

# 04 | Findings

Overall, we report 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 don't have an immediate impact but will help mitigate future vulnerabilities.

The below chart displays the findings by severity.



## Proof of Concepts

For each vulnerability we created a proof of concept to enable easy regression testing. We recommend integrating these as part of a comprehensive test suite. The proof of concept directory structure can be found in Appendix C.

A GitHub repository containing these proof of concepts can be found at https://osec.io/pocs/jet-governance.

To run a POC:

```
./run.sh <directory name>
```

For example,

```
./run.sh os-jet-adv-00
```

The relevant code for each POC can be found in `<dir>/src/main.rs`. These POCs make use of a shared framework defined in `framework/`, which is symlinked from `<dir>/src/framework.rs`.

Each proof of concept also comes with its own patch. This patch is used to enable exports from the `jet-governance` library as well as fix other prerequisite bugs. For example, we patched the conversion share calculations to demonstrate how the rounding bug (OS-JET-ADV-01) is its own separate issue.

# 05 | Vulnerabilities

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

Rating criterion can be found in Appendix E.

| ID | Severity | Status | Description |
|---|---|---|---|
| OS-JET-ADV-00 | **Critical** | **Resolved** | Easily influenceable conversion rate between shares and tokens allows a malicious user to: - Mint arbitrary amounts of vote tokens - Steal staked tokens from other users |
| OS-JET-ADV-01 | **Critical** | **Resolved** | Rounding error in converting between shares and tokens allows a malicious user to extract all of the staked tokens |
| OS-JET-ADV-02 | **High** | **Resolved** | Cached conversion rate during unbond leads to loss of funds after conversion rate decreases during unbond period |
| OS-JET-ADV-03 | **Medium** | **Resolved** | Lack of finalization check on airdrop claim allows double claiming of rewards and DOS of finalization |
| OS-JET-ADV-04 | **Medium** | **Resolved** | Truncation of overly long distribution seeds leads to denial of service on claim and close |
| OS-JET-ADV-05 | **Low** | **Resolved** | Seed collision between distribution accounts leads to denial of service |

## OS-JET-ADV-00 [Critical] [Resolved]: Conversion Rate Abuse

### Description

The use of `shares_unbonded` when performing conversion calculations allows an attacker to easily manipulate the conversion rate to their own benefit.

As seen below, the `convert_amount` function uses `self.shares_bonded` as the count for shares [0].

*staking/src/state.rs*

```rust
    pub fn convert_amount(
        &self,
        vault_amount: u64,
        amount: Amount,
    ) -> Result<FullAmount, ErrorCode> {
        if amount.value == 0 {
            msg!("the amount cannot be zero");
            return Err(ErrorCode::InvalidAmount);
        }

        let tokens = std::cmp::max(vault_amount, 1);
        let shares = std::cmp::max(self.shares_bonded, 1); // [0]

        let full_amount = FullAmount { shares, tokens };

        Ok(match amount.kind {
            AmountKind::Tokens => full_amount.with_tokens(amount.value),
            AmountKind::Shares => full_amount.with_shares(amount.value),
        })
    }
```

This share quantity is then used in the calculations for the conversion rate between tokens and shares [1].

*staking/src/state.rs*

```
    fn with_tokens(&self, tokens: u64) -> Self {
        let shares = (tokens as u128) * (self.shares as u128) /
(self.tokens as u128); // [1]
        assert!(shares < std::u64::MAX as u128);
        assert!((shares > 0 && tokens > 0) || (shares == 0 && tokens
== 0));

        let shares = shares as u64;
        Self { shares, tokens }
    }
```

This uses a linear conversion between shares and tokens. In other words, each token is worth `self.shares_bonded / staked_token_amount` shares.

However, the `self.shares_bonded` quantity can be easily changed with the `unbond` method on StakePool.

*staking/src/state.rs*

```
    pub fn unbond(&mut self, amount: &FullAmount) {
        self.shares_bonded =
self.shares_bonded.checked_sub(amount.shares).unwrap();
        self.shares_unbonded =
self.shares_unbonded.checked_add(amount.shares).unwrap();
    }
```

This method can be called by an unprivileged user, as seen in `unbond_stake_handler` [2].

*staking/src/instructions/unbond_stake.rs*

```
pub fn unbond_stake_handler(
    ctx: Context<UnbondStake>,
    _seed: u32,
    amount: Amount,
) -> ProgramResult {
    let stake_pool = &mut ctx.accounts.stake_pool;
    let stake_account = &mut ctx.accounts.stake_account;
    let unbonding_account = &mut ctx.accounts.unbonding_account;
    let clock = Clock::get()?;

    let vault_amount =
token::accessor::amount(&ctx.accounts.stake_pool_vault)?;
    let full_amount = stake_pool.convert_amount(vault_amount,
amount)?;

    stake_pool.unbond(&full_amount); // [2]
    stake_account.unbond(&full_amount)?;

    unbonding_account.stake_account = stake_account.key();
    unbonding_account.amount = full_amount;
    unbonding_account.unbonded_at = clock.unix_timestamp +
stake_pool.unbond_period;

    Ok(())
}
```

In other words, any user is able to directly affect the conversion rate between shares and tokens. As an example of potentially unexpected behavior, consider a malicious user who deposits 1000 shares and tokens (the initial conversion rate is 1 to 1). They could then unbond 999 shares, causing the conversion rate between shares and tokens to become 1 to 1000. Any future users would then be forced to use this 1 to 1000 conversion rate.

## Proof of Concept

Because conversions represent such a core aspect of the staking design, there are many ways to exploit this. In this report, we present two such example exploits.

The first exploit allows a malicious user to steal the staked tokens of another user. Consider the following series of actions:

1. Initialize two accounts: attacker and victim
2. Victim deposits 100 stake
3. Attacker deposits 1 stake

4. At this point, the pool has 101 bonded shares and 101 tokens. Now the victim tries to unbond and unbonds all their shares.
5. Now the pool only has 1 unbonded stake and 101 tokens. If the attacker unbonds, the program will think each share is worth all 101 tokens.
6. If the attacker withdraws prior to the victim, they will steal all 101 tokens. The victim will also be unable to withdraw their unbonded shares due to a lack of tokens in the vault.

The critical part of this exploit happens in step 5. Because the victim adjusted the unbonded shares in the pool, the conversion rate shifts to the attacker's advantage.

The second exploit allows a malicious user to mint unlimited vote tokens.

1. Initialize one account: attacker
2. Attacker deposits 100 stake
3. Attacker unbonds 99 stake. The pool now has 1 bonded share and 100 tokens.
4. The attacker can now mint 100 vote tokens. This is because the value of each share is calculated as 100 tokens.
5. After the unbonding period, the attacker can withdraw the unbonded staked tokens
6. The attacker gets 100 vote tokens for the price of 1 stake

The critical part of this exploit happens in step 4. Because the conversion rate was previously adjusted in step 3, the attacker is able to mint vote tokens at a much lower rate than expected.

## Remediation

From a security standpoint, the conversion rates could be made secure by using `self.shares_bonded + self.shares_unbonded` in the conversion calculations.

```
diff --git a/programs/staking/src/state.rs
b/programs/staking/src/state.rs
index 625ac3e..7e6da1f 100644
--- a/programs/staking/src/state.rs
+++ b/programs/staking/src/state.rs
@@ -70,7 +70,7 @@ impl StakePool {
         }

         let tokens = std::cmp::max(vault_amount, 1);
-        let shares = std::cmp::max(self.shares_bonded, 1);
```

```
+         let shares = std::cmp::max(self.shares_bonded +
self.shares_unbonded, 1);

          let full_amount = FullAmount { shares, tokens };
```

Although this calculation *should* never overflow, it would not be a bad idea to use `checked_add` to explicitly enforce such a critical invariant. This prevents a user from manipulating the ratio by unbonding.

Another solution would be to have the conversion rate stored as a field in the StakePool struct. This conversion rate could then be updated by a privileged user. It's important that such conversion rate updates are done atomically with any operations that affect the staked token count to avoid race conditions.

For another possible fix, see OS-JET-ADV-02.

## Patch

Resolving this issue required a more in-depth discussion with the Jet Protocol team.

The final implementation separates out the bonded and unbonding pools, treating the tokens and shares in each one as entirely independent. As a result, unbonding no longer locks in an exchange rate because tokens can be removed from the unbonding pool under the `withdraw_bonded` handler. Users are also unable to manipulate the exchange rate easily, directly mitigating the root cause of this vulnerability.

At the same time, unbonding shares will not accrue in value if the vault balance increases. This means that there still exists a pseudo-"locked-in exchange rate" against the unbonding user, maintaining the original intended economic incentive.

The root cause for this vulnerability and OS-JET-ADV-02 lies in improperly aligned economic incentives as opposed to missing security checks. These problems tend to require more work to resolve, but are often more interesting from a security standpoint.

## OS-JET-ADV-01 [Critical] [Resolved]: Conversion Rounding Error

### Description

Conversion between tokens and shares rounds the output improperly. This could allow for the extraction of the entire pool of staked tokens.

Note that this is a separate issue from OS-JET-ADV-00. Hence for these tests, we use our unbonded + bonded patch to demonstrate how the rounding bug remains exploitable.

Note how the following conversion code does not perform any rounding [0].

*staking/src/state.rs*

```
   fn with_tokens(&self, tokens: u64) -> Self {
        let shares = (tokens as u128) * (self.shares as u128) /
(self.tokens as u128); // [0]
        assert!(shares < std::u64::MAX as u128);
        assert!((shares > 0 && tokens > 0) || (shares == 0 && tokens
== 0));

        let shares = shares as u64;
        Self { shares, tokens }
    }

    fn with_shares(&self, shares: u64) -> Self {
        let tokens = (self.tokens as u128) * (shares as u128) /
(self.shares as u128); // [0]
        assert!(tokens < std::u64::MAX as u128);
        assert!((shares > 0 && tokens > 0) || (shares == 0 && tokens
== 0));

        let tokens = tokens as u64;
        Self { shares, tokens }
    }
```

This means that the correctness of this code depends on floor division always being worse for the user. A simple counterexample can be found in the `mint_votes` handler.

*staking/src/instructions/mint_votes.rs*

```
pub fn mint_votes(&mut self, amount: &FullAmount) -> Result<(),
ErrorCode> {
    self.minted_votes =
self.minted_votes.checked_add(amount.tokens).unwrap();

    let minted_vote_amount = amount.with_tokens(self.minted_votes);

    if minted_vote_amount.shares > self.shares { // [1]
        let max_amount = amount.with_shares(self.shares);
        msg!(
            "insufficient stake for votes: requested={},
available={}",
            minted_vote_amount.tokens,
            max_amount.tokens
        );
        return Err(ErrorCode::InsufficientStake);
    }

    Ok(())
}
```

In this check, the number of shares required should always be rounded up [2]. Otherwise, the user might be able to mint tokens with one less share than they should.

## Proof of Concept

An important question to answer with rounding bugs is if the rounding bug can be repeatedly triggered. If not, the severity of the vulnerability would be reduced.

To answer this question, we first did some napkin math. There are two types of operations that we can do as an unprivileged user to influence the share and token count.

1. `add_stake`: This adds both stake and tokens
2. Directly send tokens to the vault stake account

We can easily simulate the effects of these two operations on the share and token count. We can also set sane limits on both operations. For example, it probably never makes sense to transfer 100 million tokens to try and exploit a rounding bug. It's likely that a smaller token count would transfer, and such capital would be hard to realistically come by (flash loans don't apply here because of delayed unbond).

With these assumptions as our framework, we wrote up a quick test script in Javascript. This script can be found in Appendix D. This test script quickly revealed that this rounding behavior was indeed repeatedly exploitable.

```
profit:  200
cost:  400
```

Having this confirmation, we then wrote up a full proof of concept against the onchain program to confirm exploitability.

As an aside, while rounding errors might seem uneconomical to exploit compared to Solana's transaction cost, there are a number of tricks we can use.

1. Multiple instructions can be packed into one transaction
2. An onchain program can be used to repeatedly call the target

In practice, this means that many rounding exploits are actually quite serious issues and should be treated as such.

## Remediation

The code should properly round amounts against the user.

This rounding could be implemented by passing in the intended rounding direction into the conversion function.

As a reference, see SPL token-swap's pool_tokens_to_trading_tokens [2].

https://github.com/solana-labs/solana-program-library/blob/master/token-swap/program/src/curve/constant_product.rs#L67-L68

```
pub fn pool_tokens_to_trading_tokens(
    pool_tokens: u128,
    pool_token_supply: u128,
    swap_token_a_amount: u128,
    swap_token_b_amount: u128,
    round_direction: RoundDirection,
) -> Option<TradingTokenResult> {
    let mut token_a_amount = pool_tokens
        .checked_mul(swap_token_a_amount)?
        .checked_div(pool_token_supply)?;
    let mut token_b_amount = pool_tokens
        .checked_mul(swap_token_b_amount)?
        .checked_div(pool_token_supply)?;
    let (token_a_amount, token_b_amount) = match round_direction { // [2]
        RoundDirection::Floor => (token_a_amount, token_b_amount),
        RoundDirection::Ceiling => {
```

<u>**Patch**</u>

Rounding direction is now passed into `with_tokens` and `with_shares`.

```
    pub fn with_tokens(&self, rounding: Rounding, token_amount: u64)
-> Self {
        // Given x = a / b, we round up by introducing c = (b - 1)
and calculating
        // x = (c + a) / b.
        // If a % b = 0, c / b = 0, and introduces no rounding.
        // If a % b > 0, c / b = 1, and rounds up by adding 1 to x.
        let round_amount = match rounding {
            Rounding::Up if token_amount > 0 => self.all_tokens as
u128 - 1,
            _ => 0,
        };
```

## OS-JET-ADV-02 [High] [Resolved]: Unbond Cancel Risk Hedging

### Description

The use of `unbond_stake` along with `cancel_unbond` enables an attacker to hedge risk if the conversion rate between shares and tokens is volatile, potentially leading to a loss of funds for other users.

As seen below, `cancel_unbond` which calls `rebond` on both `StakePool` and `StakeAccount` only takes into account the number of unbonded shares.

*staking/src/state.rs*

```
    pub fn withdraw(&mut self, amount: &FullAmount) {
        self.shares_unbonded =
 self.shares_unbonded.checked_sub(amount.shares).unwrap();
    }

    pub fn unbond(&mut self, amount: &FullAmount) {
        self.shares_bonded =
 self.shares_bonded.checked_sub(amount.shares).unwrap();
        self.shares_unbonded =
 self.shares_unbonded.checked_add(amount.shares).unwrap();
    }

    pub fn rebond(&mut self, amount: &FullAmount) {
        self.withdraw(amount);
        self.deposit(amount);
    }
```

*staking/src/state.rs*

```
    pub fn deposit(&mut self, amount: &FullAmount) {
        self.shares = self.shares.checked_add(amount.shares).unwrap();
    }

    pub fn rebond(&mut self, amount: &FullAmount) {
        self.withdraw_unbonded(amount);
        self.deposit(amount);
    }

    pub fn withdraw_unbonded(&mut self, amount: &FullAmount) {
        self.unbonding = self.unbonding.checked_sub(amount.shares).unwrap();
    }
```

This means that a user could unbond immediately and nullify any risk of the rate of tokens/shares changing. If the conversion rate increases they can simply cancel the unbond. If the conversion rate decreases they can then redeem their unbond account for the amount of tokens the shares were originally worth through `withdraw_unbonded`.

In other words, there is no reason for a user *not* to immediately unbond all their stake.

## Proof of Concept

Consider the following example.

1. Both the victim and attacker deposit 100 tokens at a rate of 1 share : 1 token
2. The attacker unbonds their 100 shares at the 1:1 rate
3. The admin of the stake pool withdraws 100 tokens through `withdraw_bonded`, reducing the rate of tokens: shares to 1:2
4. The attacker withdraws their 100 unbonded shares at the 1:1 rate
5. The stake pool now has 0 tokens for the victim to withdraw, so their 100 shares are worth 0.

## Remediation

The unbond should recalculate the conversion rate and take the minimum rate for the user.

For example, if the rate of tokens to shares increased after the user unbonded, allow them to complete their unbond at the original rate. Otherwise, recalculate the conversion rate and complete the unbond at the new rate.

Another possible solution would be to store the amount of unbonding tokens in the stake pool as a field on the stake pool alongside `shares_unbonded`. Conversions should then be calculated between `shares_bonded` and the balance of the stake pool vault minus tokens being unbonded. In other words, conversion calculations should only be done between bonded shares and tokens. Unbonded shares and tokens in essence “disappear”.

Note that this would also work as a patch for OS-JET-ADV-00.

### Patch

This was resolved in the same patch as for OS-JET-ADV-00. The use of separate token pools removes the exchange rate lock-in design which existed previously, preventing such economic incentive abuse.

## OS-JET-ADV-03 [Medium] [Resolved]: Lack of finalized check on Airdrop claim

### Description

Airdrops can be claimed before they are finalized due to lack of checks in `airdrop_claim_handler`. An attacker could abuse this to claim the airdrop multiple times, as well as prevent airdrop finalization.

Note the lack of a finalization check in the claim handler.

*rewards/src/instructions/airdrop_claim.rs*

```
    pub fn claim(&mut self, recipient: &Pubkey) -> Result<u64,
ErrorCode> {
        let target = self.target_info_mut();
        let entry = target.get_recipient_mut(recipient)?;
        let amount = entry.amount;

        entry.amount = 0;
        target.reward_total =
target.reward_total.checked_sub(amount).unwrap();

        Ok(amount)
    }
```

Users can also be added multiple times due to the lack of a strict equality test in `add_recipients`, <= instead of < . This issue was also identified in the internal audit.

*rewards/state/airdrop.rs*

```
        // Make sure the list of recipients is SORTED
        let range = &target.recipients[start_idx as usize -
1..target.recipients_total as usize];
        let is_sorted = range.windows(2).all(|i| i[0].recipient <=
i[1].recipient);

        if !is_sorted {
            return Err(ErrorCode::RecipientsNotSorted);
        }
```

## Proof of Concept

To claim an airdrop rewards multiple times, consider the following scenario:

1. The airdrop authority provides tokens to the airdrop.
2. The airdrop adds the attacker to the airdrop in one batch.
3. The attacker claims the airdrop due to a lack of finalization check.
4. The airdrop adds the attacker to the airdrop again.
5. The attacker can then claim the airdrop again due to a different binary search order.

The critical step for this exploit happens in step 5. The award claim will look for a user with the `get_recipient_mut` function [0].

*rewards/src/instructions/airdrop_claim.rs*

```
    pub fn claim(&mut self, recipient: &Pubkey) -> Result<u64,
ErrorCode> {
        let target = self.target_info_mut();
        let entry = target.get_recipient_mut(recipient)?; // [0]
        let amount = entry.amount;

        entry.amount = 0;
        target.reward_total =
target.reward_total.checked_sub(amount).unwrap();

        Ok(amount)
    }
```

`get_recipient_mut` is implemented as a binary search on the underlying recipient array [1]. However, because this array is mutated by step 4, the search will end up selecting a different recipient object. This means that the award amount will be non-zero, allowing the attacker to claim again.

*rewards/src/state/airdrop.rs*

```
    fn get_recipient_mut(&mut self, recipient: &Pubkey) ->
Result<&mut AirdropTarget, ErrorCode> {
        let recipients = &mut self.recipients[..self.recipients_total
as usize];

        let found = recipients
            .binary_search_by_key(recipient, |r: &AirdropTarget|
r.recipient) // [1]
            .map_err(|_| ErrorCode::RecipientNotFound)?;

        Ok(&mut recipients[found])
    }
```

To prevent airdrop finalization, consider the following scenario:

1. The airdrop authority provides tokens to the airdrop.
2. The airdrop batches the recipients and adds recipients in batches with the `airdrop_add_recipients` handler.
3. When the attacker is added as a recipient, the attacker can immediately claim their reward prior to finalization.

Because the claim handler is missing a finalized check, this will allow an attacker to immediately withdraw their rewards. At this point, the token balance of the vault will be less than the initial amount.

Thus, when the airdrop authority tries to finalize the airdrop, the finalization condition will fail [0].

*rewards/src/state/airdrop.rs*

```
    pub fn finalize(&mut self, vault_balance: u64) -> Result<(),
ErrorCode> {
        let target = self.target_info_mut();

        if vault_balance < target.reward_total { // [0]
            return Err(ErrorCode::AirdropInsufficientRewardBalance);
        }

        target.finalized = 1;
        Ok(())
    }
```

**Remediation**

Return an error in `airdrop_claim_handler` if the airdrop is not finalized

**Patch**

An additional check was introduced in `Airdrop::claim`.

```
    pub fn claim(&mut self, recipient: &Pubkey) -> Result<u64,
ErrorCode> {
        let target = self.target_info_mut();

        if target.finalized != 1 {
            msg!("cannot claim from an unfinalized airdrop");
            return Err(ErrorCode::AirdropNotFinal);
        }

        let entry = target.get_recipient_mut(recipient)?;
        let amount = entry.amount;

        entry.amount = 0;
        target.reward_total =
target.reward_total.checked_sub(amount).unwrap();

        Ok(amount)
    }
```

## OS-JET-ADV-04 [Medium] [Resolved]: Seed Truncation

### Description

The seed used for distributions and stake pools can become desynchronized with the stored seed_len, leading to a denial of service condition for seeds with length greater than the internal seed buffer length, or in the current implementation, `30`. If an attacker could influence the seed, they could prevent the distribution from being claimed or closed.

Below we use `Distribution` as an example to illustrate this vulnerability.

Note how both the seed itself and a separate length is stored.

*rewards/src/state/distribution.rs*

```
pub struct Distribution {
    /// The address of this distribution account
    pub address: Pubkey,

    /// The authority that can manage this distribution.
    pub authority: Pubkey,

    /// The account with the tokens to be distributed
    pub vault: Pubkey,

    /// The seed for the address
    pub seed: [u8; 30],

    /// The length of the seed string
    pub seed_len: u8,
```

However, the write call in `distribution_create_handler` will silently drop the extra data if the size of the input exceeds the underlying buffer's length.

*rewards/src/instructions/distribution_create.rs*

```
pub fn distribution_create_handler(
    ctx: Context<DistributionCreate>,
    params: DistributionCreateParams,
) -> ProgramResult {
    let distribution = &mut ctx.accounts.distribution;

    distribution.address = distribution.key();
    distribution.seed.as_mut().write(params.seed.as_bytes())?;
    distribution.seed_len = params.seed.len() as u8;
```

On the other hand, the `seed_len` property will be set to the original input string's length.

This will cause an out of bounds access in `signer_seeds`, preventing the distribution from signing CPI calls. More specifically, note that `self.seed` is a buffer of length 30, but `self.seed_len` could be greater than 30.

*rewards/src/state/distribution.rs*

```
impl Distribution {
    pub fn signer_seeds(&self) -> [&[u8]; 2] {
        [
            &self.seed[..self.seed_len as usize],
            self.bump_seed.as_ref(),
        ]
    }
}
```

In other words, the distribution can neither be closed or claimed, because both functions involve CPI [0]. For brevity we only show closing accounts below.

*rewards/src/instructions/distribution_close.rs*

```
pub fn distribution_close_handler(ctx: Context<DistributionClose>) ->
ProgramResult {
    let distribution = &ctx.accounts.distribution;
    let clock = Clock::get()?;

    if distribution.end_at > (clock.unix_timestamp as u64) {
        msg!("distribution has not ended yet");
        return Err(ErrorCode::DistributionNotEnded.into());
    }

    token::close_account(
        ctx.accounts
            .close_vault_context()
            .with_signer(&[&distribution.signer_seeds()]), // [0]
    )?;

    Ok(())
}
```

This vulnerability would also have been mitigated by OS-JET-SUG-04, by handling the return value of write.

## Proof of Concept

Note that to demonstrate this vulnerability, we include a patch for the award struct. This is because the award struct originally uses String to store the seed which is dynamically sized, and does not serialize properly with Anchor.

To demonstrate this exploit, we simply use a seed with length one greater than the internal buffer.

## Remediation

Ensure that the input string length does not exceed the length of the underlying buffer.

## Patch

The use of `write_all` ensures that the entire input string gets written to the underlying buffer.

```
pub fn distribution_create_handler(
    ctx: Context<DistributionCreate>,
    params: DistributionCreateParams,
) -> ProgramResult {
    let distribution = &mut ctx.accounts.distribution;

    distribution.address = distribution.key();
    distribution
        .seed
        .as_mut()
        .write_all(params.seed.as_bytes())?;
```

## OS-JET-ADV-05 [Low] [Resolved]: Distribution Account Collision

### Description

The lack of a unique seed for the Distribution struct could potentially lead to a denial of service condition if a predictable seed is used.

In `DistributionCreate`, the `distribution` account is keyed entirely on the seed parameter.

*rewards/src/instructions/distribution_create.rs*

```
pub struct DistributionCreate<'info> {
    /// The account to store the distribution info
    #[account(init,
              seeds = [params.seed.as_bytes()],
              bump,
              payer = payer_rent)]
    pub distribution: Account<'info, Distribution>,
```

Consider this in contrast to `AwardCreate` which also uses `params.stake_account` as seed data.

*rewards/src/instructions/award_create.rs*

```
    /// The award being created
    #[account(init,
              seeds = [
                  params.stake_account.as_ref(),
                  params.seed.as_bytes(),
              ],
              bump,
              payer = payer_rent
    )]
    pub award: Account<'info, Award>,
```

If two users attempt to create an account with the same seed, Anchor will attempt to create the same account causing the second attempt to fail.

A similar theoretical risk occurs for accounts of different types. For example with `AwardCreate`, if `params.stake_account` can be expressed as a UTF-8 string, then a malicious attacker could choose the distribution seed to overlap with the chosen Award. In reality however, the chance of a collision is extremely minimal (experimentally approximately 1/100000000) that this isn't a very serious concern.

## Proof of Concept

To demonstrate this vulnerability, we attempt to create a distribution with the same seed as both a victim and attacker.

## Remediation

Distribution should key the distribution object on user specific data, such as `target_account` [0].

*rewards/src/instructions/distribution_create.rs*

```
#[derive(AnchorDeserialize, AnchorSerialize)]
pub struct DistributionCreateParams {
    /// The seed to create the address for the distribution
    pub seed: String,

    /// The authority allowed to manage the distribution
    pub authority: Pubkey,

    /// The token account to send the distributed tokens to
    pub target_account: Pubkey, // [0]
```

The seeds for different reward structures should also be appended with a unique identifier. For example, see this implementation from Drift Protocol.

https://github.com/drift-labs/protocol-v1/blob/master/programs/clearing_house/src/context.rs#L27

```
   #[account(
       init,
       seeds = [b"clearing_house".as_ref()],
       bump = clearing_house_nonce,
       payer = admin
```

```
    )]
    pub state: Box<Account<'info, State>>,
    pub collateral_mint: Box<Account<'info, Mint>>,
    #[account(
        init,
        seeds = [b"collateral_vault".as_ref()],
        bump = collateral_vault_nonce,
        payer = admin,
        token::mint = collateral_mint,
        token::authority = collateral_vault_authority
    )]
    pub collateral_vault: Box<Account<'info, TokenAccount>>,
    pub collateral_vault_authority: AccountInfo<'info>,
    #[account(
        init,
        seeds = [b"insurance_vault".as_ref()],
        bump = insurance_vault_nonce,
        payer = admin,
        token::mint = collateral_mint,
        token::authority = insurance_vault_authority
    )]
```

**Patch**

A seed prefix was introduced for the `Distribution` struct.

```
     /// The account to store the distribution info
     #[account(init,
               seeds = [
                   b"distribution".as_ref(),
                   params.seed.as_bytes()
               ],
               bump,
               payer = payer_rent)]
     pub distribution: Account<'info, Distribution>,
```

# 06 | General Findings

Here we present a discussion of general findings during our audit. While these findings do not present an immediate security impact, they do represent antipatterns and could introduce a vulnerability in the future.

| ID | Description |
|---|---|
| OS-JET-SUG-00 | Check parameters for the rewards/ program on creation |
| OS-JET-SUG-01 | Ensure airdrop finalize can only be called once to avoid potential invalid state transitions |
| OS-JET-SUG-02 | Explicate airdrop finalization state |
| OS-JET-SUG-03 | Explicate airdrop numerical constants |
| OS-JET-SUG-04 | Unhandled write return value truncates strings |
| OS-JET-SUG-05 | Explicitly enforce distribution time frame invariants |
| OS-JET-SUG-06 | Avoid potential pitfall with `minted_collateral` field |

## OS-JET-SUG-00: *_create Validation

### Description

Parameters are not properly validated in any of the creation methods. While this does not pose an immediate security concern due to the privileged nature of these creation methods, a lack of checks could introduce unexpected behavior.

In all of the reward creation handlers, input parameters are not properly validated.

- `airdrop_create_handler`: *rewards/src/instructions/airdrop_create.rs*
- `distribution_create_handler`: *rewards/src/instructions/distribution_create.rs*
- `award_create_handler`: *rewards/src/instructions/award_create.rs*

For brevity, we only include the airdrop creation handler below as an example.

*rewards/src/instructions/airdrop_create.rs*

```rust
pub fn airdrop_create_handler(
    ctx: Context<AirdropCreate>,
    params: AirdropCreateParams,
) -> ProgramResult {
    let mut airdrop = ctx.accounts.airdrop.load_init()?;

    airdrop.address = ctx.accounts.airdrop.key();
    airdrop.authority = ctx.accounts.authority.key();
    airdrop.reward_vault = ctx.accounts.reward_vault.key();
    airdrop.vault_bump[0] = *ctx.bumps.get("reward_vault").unwrap();

    airdrop.expire_at = params.expire_at;
    airdrop.stake_pool = params.stake_pool;

    airdrop.flags = params.flags;

    airdrop
        .short_desc
        .as_mut()
        .write(params.short_desc.as_bytes())?;

    Ok(())
```

Note how no input validation is done. For example, the expiration date could be in the past or extremely far in the future.

## Remediation

We recommend validation for the following fields.

`airdrop_create_handler`

1. `flags` should not allow arbitrary values.
2. `expire_at` should be in the future.
3. `expire_at` should not be too far into the future, for example, more than 100 years. This will help mitigate arithmetic overflow errors.

`reward_create_handler` and `distribution_create_handler`

1. `begin_at` should be in the future.
2. `end_at` should be strictly larger than `begin_at`.
3. `target_amount` should be greater than 0
4. `begin_at` and `end_at` should not be too far into the future, for example, more than 100 years. This will help mitigate arithmetic overflow errors.

## OS-JET-SUG-01: Airdrop Duplicate Finalize

### Description

Airdrop finalization changes the state of the airdrop to finalized, ensuring that no future users can be added to the airdrop.

While this call represents a state transition, it does not make any checks about the initial state of the airdrop. Note the lack of a check against `target.finalized`.

*rewards/src/instructions/airdrop_finalize.rs*

```
    pub fn finalize(&mut self, vault_balance: u64) -> Result<(),
ErrorCode> {
        let target = self.target_info_mut();

        if vault_balance < target.reward_total {
            return Err(ErrorCode::AirdropInsufficientRewardBalance);
        }

        target.finalized = 1;
        Ok(())
    }
```

State transitions are a likely source of future vulnerabilities, and the existing behavior should be documented.

### Remediation

Explicitly check that the target is not finalized yet prior to finalization.

## OS-JET-SUG-02: Explicate Airdrop State

### Description

Airdrop finalization state is currently stored as a `u64`.

*rewards/src/state/airdrop.rs*

```rust
#[repr(C)]
#[assert_size(400024)]
#[derive(Clone, Copy)]
pub struct AirdropTargetInfo {
    /// The total amount of reward tokens that are claimable by recipients
    pub reward_total: u64,

    /// The total number of airdrop recipients
    pub recipients_total: u64,

    /// Marker to indicate when the airdrop has been finalized
    /// from further edits
    pub finalized: u64,

    /// List of airdrop recipients that can claim tokens
    pub recipients: [AirdropTarget; 10000],
}
```

If the finalization is meant as a boolean state (finalized or not), the type should be changed to a boolean. Otherwise, the type should be an enum. This helps avoid issues with unexpected state transitions, a very common source of vulnerabilities.

### Remediation

Change the type of `finalized` to either a boolean or enum value. In the case of an enum value, it would also be more declarative to change the field name to `state`.

## OS-JET-SUG-03: Airdrop Constants

### Description

The use of undocumented constants potentially raises a maintenance concern.

*rewards/src/state/airdrop.rs*

```
    /// A short descriptive text for the airdrop
    pub short_desc: [u8; 31],

    /// The bump seed for the reward vault
    pub vault_bump: [u8; 1],

    /// Storage space for the list of airdrop recipients
    pub target_info: [u8; 400024],
```

It is not clear that `400024` must be kept in sync with the size of AirdropTargetInfo. A potential desync in these sizes could raise a security concern.

*rewards/src/state/airdrop.rs*

```
#[repr(C)]
#[assert_size(400024)]
#[derive(Clone, Copy)]
pub struct AirdropTargetInfo {
```

### Remediation

Use `std::mem::size_of<AirdropTargetInfo>()` to explicitly declare where the `400024` came from. Note that because `size_of` is a const function, it can be used inline in the struct declaration.

## OS-JET-SUG-04: Unhandled Write Value

### Description

The return value of the write call is not properly handled in multiple places. This might cause unexpected truncation of the input string.

For example, see `init_pool_handler` [0].
*staking/src/instructions/init_pool.rs*

```
pub fn init_pool_handler(
    ctx: Context<InitPool>,
    seed: String,
    config: PoolConfig,
) -> ProgramResult {
    let stake_pool = &mut ctx.accounts.stake_pool;

    stake_pool.authority = ctx.accounts.authority.key();
    stake_pool.token_mint = ctx.accounts.token_mint.key();
    stake_pool.stake_pool_vault =
ctx.accounts.stake_pool_vault.key();
    stake_pool.stake_vote_mint = ctx.accounts.stake_vote_mint.key();

    stake_pool.bump_seed[0] = *ctx.bumps.get("stake_pool").unwrap();
    stake_pool.seed.as_mut().write(seed.as_bytes())?; // [0]
    stake_pool.seed_len = seed.len() as u8;

    stake_pool.unbond_period = config.unbond_period as i64;

    Ok(())
}
```

The `stake_pool.seed` field is defined as a u8 array [1].
*staking/src/state.rs*

```
#[account]
#[derive(Default)]
pub struct StakePool {
    /// The authority allowed to withdraw the staked tokens
    pub authority: Pubkey,
    /// The seed used to generate the pool address
    pub seed: [u8; 30], // [1]
    pub seed_len: u8,
    pub bump_seed: [u8; 1],
```

According to the Rust documentation for `std::io::Write`.

```
fn write(&mut self, buf: &[u8]) -> Result<usize>
```

> *Write a buffer into this writer, returning how many bytes were written.*
>
> *This function will attempt to write the entire contents of buf, but the entire write might not succeed, or the write may also generate an error. A call to write represents at most one attempt to write to any wrapped object.*

Passing in a string of more than 30 characters as the seed will cause the remaining characters to be silently truncated.

`write` is improperly used in the below places:

- `airdrop_create_handler`: *rewards/src/instructions/airdrop_create.rs*
- `distribution_create_handler`: *rewards/src/instructions/distribution_create.rs*
- `init_pool_handler`: *staking/src/instructions/init_pool.rs*

## Remediation

Consider:

- Ensuring that `seed.len()` is validated against `stake_pool.seed.len()`
- Assert that the return value of the write call is equal to the string length
- Use `write_all` instead of `write`

## OS-JET-SUG-05: Distribution Timeframe Invariants

### Description

A currently unexploitable arithmetic underflow could be further mitigated by enforcing stricter invariants on the input data.

We also note that this was a discrepancy with the internal audit. Specifically, the internal audit asserted that

> In the same function above, the **remaining** variable is checked to prevent a saturating sub, which guarantees that the remaining arithmetic in the function is safe. More specifically, **range** can never be < **remaining**.

The saturating sub does not guarantee that range can never be less than remaining. This is because `begin_at` might be larger than the current timestamp.

*rewards/src/state/distribution.rs*

```
    fn distributed_amount_linear(&self, timestamp: u64) -> u64 {
        let range = std::cmp::max(1, self.end_at - self.begin_at) as
u128;
        let remaining = self.end_at.saturating_sub(timestamp) as
u128;
        let target_amount = self.target_amount as u128;

        let distributed = target_amount - (remaining * target_amount)
/ range;
        assert!(distributed < std::u64::MAX as u128); // [0]

        distributed as u64
    }
```

When the distribution has not started, remaining will be larger than range leading to an arithmetic underflow. This results in an assertion failure because `distributed` will result in a very large number (usually far greater than `std::u64::MAX`).

More strictly, we will demonstrate that the assertion against `std::u64::MAX` is enough to guarantee that the underflow will never be exploitable. In other words, we will show that if an underflow happens, `distributed` must be larger than `std::u64::MAX`. This will cause the assertion to trigger, leading to an unexploitable crash.

$$0 \leq target\_amount < 2^{64}$$
$$0 \leq remaining < 2^{64}$$
$$range \geq 1$$

$$target\_amount - (remaining * target\_amount)/range$$
$$\geq 2^{128} - target\_amount + remaining * target\_amount \text{ (due to underflow)}$$
$$= 2^{128} - remaining * (target\_amount - 1)$$
$$\geq 3 * 2^{64} - 2$$
$$> 2^{64} - 1$$

**Remediation**

If the distribution has started, explicitly enforce that `range` should always be less than or equal to `remaining` with an assertion.

If the distribution has not started, return 0 or present a better error message to the user.

## OS-JET-SUG-06: Potential minted_collateral oversight

### Description

Currently, the `StakePool` struct contains a field `minted_collateral` which is never initialized or modified. This field is currently only used in the `unbond` function.

*staking/src/state.rs*

```rust
#[account]
#[derive(Default)]
pub struct StakeAccount {
   /// The account that has ownership over this stake
   pub owner: Pubkey,

   /// The pool this account is associated with
   pub stake_pool: Pubkey,

   /// The stake balance (in share units)
   pub shares: u64,

   /// The token balance locked by existence of voting tokens
   pub minted_votes: u64,

   /// The stake balance locked by existence of collateral tokens
   pub minted_collateral: u64,

   /// The total staked tokens currently unbonding so as to be withdrawn in the
future
   pub unbonding: u64,
}
```

This field is checked in the `unbond` function, where the usage implies that collateral is meant to be 1:1 with shares.

*staking/src/state.rs*

```rust
   pub fn unbond(&mut self, amount: &FullAmount) -> Result<(), ErrorCode> {
       if self.shares < amount.shares {
           return Err(ErrorCode::InsufficientStake);
       }

       self.shares = self.shares.checked_sub(amount.shares).unwrap();
```

```
        self.unbonding = self.unbonding.checked_add(amount.shares).unwrap();

        let minted_vote_amount = amount.with_tokens(self.minted_votes);
        if minted_vote_amount.shares > self.shares {
            return Err(ErrorCode::VotesLocked);
        }

        if self.minted_collateral > self.shares {
            return Err(ErrorCode::CollateralLocked);
        }

        Ok(())
    }
```

Since `minted_vote_amount` is checked independently from `minted_collateral`, an attacker could mint votes and collateral at the same time and then unbond their shares, leaving insufficient shares to back both the votes and collateral.

### Remediation

Instead of having two independent if statements, they should be combined into a single if statement that checks if the sum of `minted_vote_amount` and `minted_collateral` is greater than the final number of shares owned by the user.

# 07 | Appendix

## Appendix A: Program Files

Below are the files in scope for this audit and their corresponding sha256 hashes.

```
auth
  Cargo.toml                    7f6ff71d29e6c651e8cc041aef9557c2cbff5f102b647a293b0666af5bb2ab01
  Xargo.toml                    815f2dfb6197712a703a8e1f75b03c6991721e9eb7c40dfaec8b0b49da4aa629
  src
    lib.rs                      592efa722b80b205e96d823dbbc2a0b5687990df1114cc32a4ef469f1a25d3ba
rewards
  Cargo.toml                    ebfe62a749a730ebb2c0640d9184a4b943d8cc0d1e3970ceb6b8588cf766dde7
  Xargo.toml                    815f2dfb6197712a703a8e1f75b03c6991721e9eb7c40dfaec8b0b49da4aa629
  src
    instructions
      award_revoke.rs           c8f625f548a62260b6f1399978e41c9eecda410ba9cae636f4945b4e3f7caf00
      distribution_create.rs    7c317a1f02f43ad047f2a046efc9fc706cb684356cca574581d43e70a4abbaae
      airdrop_add_recipients.rs f10ddddd678399dcba495cc60345f66902b3c5604534f8ae6a88fe9319e03bd3
      award_close.rs            9020422305dbbabec20de6fb537dead2141a370d1204ffafff2c3fb27814fa66
      airdrop_close.rs          46be4df49bdb0429378ac263d6fa5a6e6a7d97503c18c51ca8e978fde1cfba4e
      distribution_release.rs   5f66c23a7304e13d0eaf1ba8d31402d49aba49129118e0a6bdf9b1d1d9f144d0
      airdrop_finalize.rs       04c56eb15b13039b12c56e36f66eaa6625821819adf1f07d8ce1dbb95b33beee
      award_create.rs           a6f1fe00784d29b5980a5b0167a30df96432e8be95a92ce9cfacdf0373968fe0
      award_release.rs          7df18e4bc8502a1233b7d3fc9f81ed01d57a9abaec938bed8fc85b4e26f71c47
      airdrop_create.rs         c00c52bb1042f9a354e58c624162a65d4e5813333effdde51f2d5fd88e72a74f
      airdrop_claim.rs          1e791fc5914bb9b0f709e2208af05099a75281ac5fa407c010e6174009609a1e
      distribution_close.rs     b2b9f3df57fbd378b8b7beebb9c66e39a290872ac1dd3fc09cc21d00b619929e
    instructions.rs             ff55aedbe906acaaf05ee0c78c7cf4cd5e3dcee471b1476b8581c09cb43440f4
    lib.rs                      debc1a2fa419f118e7a0e8ee635548382c04a168ec74532e5d077c932e440b7f
    state
      distribution.rs           6d2449c50aa77187da699376f39d9353abb7c32e17c69a4a10a2d754b5c41828
      award.rs                  21ed020c9030689d89d7c8cf0e456c26732f440356502fc5c8b119e8a04e8d61
      airdrop.rs                585cd8c1d7eb2c01384a6743f29a1b00ab5802419a7e826f62aaaca63ece3f13
    state.rs                    fcf194776a6b124a6aab4a23962d6b9f39ad2088c273d596203ac924cd8d7397
staking
  Cargo.toml                    64abd2aa9728aac6dbb95740a361f814a2203f858eb64e97f8860ef1c010a9d3
  Xargo.toml                    815f2dfb6197712a703a8e1f75b03c6991721e9eb7c40dfaec8b0b49da4aa629
  src
    instructions
      add_stake.rs              cbb27e9661a0910b6e7cad1775b1eb9728b476500ed5deffc6c8ccf9ba5ba576
      unbond_stake.rs           0067f6059c3027b0515aa81a20168a591175de9fda9b01d1fafd02595081762c
      init_stake_account.rs     389c3e1e985f45100cda12ec292b7e00352e3783fc40e4c6d9ee0340975a9ffb
      close_stake_account.rs    32e1b03e61b477fc05c605279a8eaac1948e1fd397f0f36813f26c4fbf87c0af
      cancel_unbond.rs          3d1d4e161a2003f70a29f055fcfdcd6e13d4ca7dc670870d3696ae5bf95c5946
      withdraw_unbonded.rs      1a96c995e753da012c1cf813cc6099990cf05e607ba933eb8c878e2647d9e236
      mint_votes.rs             032b345fd6ca35238b16f3ea1cfff7a26537163707fa7e47f93f6969849e76a4
      init_pool.rs              7ea1b5b84ec5ef1654e7578daf6f6f0bf2ab38c0e7e3b78fcc30a9077fb695eb
      burn_votes.rs             fdb5b53fc0751ccf1e240d2af3b70a3aebb54f6dde6a219f576e3886db72eb21
      withdraw_bonded.rs        771815e378e8e3c47a937a9c6ff93c79d77ebd3906af65c28af65de57e8b1a13
    instructions.rs             db24aa43dec4e3f3e6871492e8254a2cde3db48bd35fe8cb2c13d7aabff53047
    lib.rs                      80c15c23f0d31104b052eed8596e85f9e500b154cc66cabb79a3a6f1bfe7261a

    state.rs                    2cee822224d8499ab4fa8ee46c64a37c6b330f7e041dee590177ab27efdfee2d
```

## Appendix B: Implementation Security Checklist

### **Unsafe arithmetic**

| | |
|---|---|
| Integer underflows or overflows | Unconstrained input sizes could lead to integer over or underflows, causing potentially unexpected behavior. Ensure that for unchecked arithmetic, all integers are properly bounded. |
| Rounding | Rounding should always be done against the user to avoid potentially exploitable off-by-one vulnerabilities. |
| Conversions | Rust `as` conversions can cause truncation if the source value does not fit into the destination type. While this is not undefined behavior, such truncation could still lead to unexpected behavior by the program. |

### **Account security**

| | |
|---|---|
| Account Ownership | Account ownership should be properly checked to avoid type confusion attacks. For Anchor, the safety of unchecked accounts should be clearly justified and immediately obvious. |
| Accounts | For non-Anchor programs, the type of the account should be explicitly validated to avoid type confusion attacks. |
| Signer Checks | Privileged operations should ensure that the operation is signed by the correct accounts. |
| PDA Seeds | PDA seeds are uniquely chosen to differentiate between different object classes, avoiding collision. |

### **Input Validation**

| | |
|---|---|
| Timestamps | Timestamp inputs should be properly validated against the current clock time. Timestamps which are meant to be in the future should be explicitly validated so. |
| Numbers | Sane limits should be put on numerical input data to mitigate the risk of unexpected over and underflows. Input data should be constrained to the smallest size type possible, and upcasted for unchecked arithmetic. |
| Strings | Strings should have sane size restrictions to prevent denial of service conditions |

| | |
|---|---|
| Internal State | If there is internal state, ensure that there is explicit validation on the input account's state before engaging in any state transitions. For example, only open accounts should be eligible for closing. |

## Miscellaneous

| | |
|---|---|
| Libraries | Out of date libraries should not include any publicly disclosed vulnerabilities |
| Clippy | `cargo clippy` is an effective linter to detect potential anti-practices. |

# Appendix C: Proof of Concepts

## Below are the provided proof of concept files and their sha256 hashes.

```
README.md                       778abc69df9adbcde53ab324bc445d5c220033cd7d29cd1b656f7d4eb9bbeeb0
build.sh                        e9c44e3a3eb27da3cf11e0e8ee04d19340025499ccce0eab939d7c90e56cdd21
run.sh                          9e97d8e1382c63593bb6c6d511b4e118bfd8dd7ec1ac61ddfc9f6f62457eb858
framework
  framework.rs                  086f29b0fc1c5c030ebddb4cc7913b2614dca119e5df18b98ac04dd788170b3e
  nop.so                        0987c0494e9e505ff9584279d07887234b322077a5bc8857b0bfe7d0989be65f
os-jet-adv-00
  Cargo.lock                    0744c262252c23021fcb97451942268cf21c0b0baa5756feb4d16b7ace2b1587
  Cargo.toml                    5bb6013a82e8b3a239d8efaed7e21d57c46862e7b93cf275a85f397e6b73bb05
  patch                         4b2dbdf3ffc65246feba91318ae6bdc4fbbab0b8b30856276c072e016c9bcfc6
  src
    framework.rs -> ../../framework/framework.rs
    main.rs                     7455929b1872daaf667f7d07bff279306d8d429f6a4348a24a9e747736656675
os-jet-adv-01
  Cargo.lock                    0744c262252c23021fcb97451942268cf21c0b0baa5756feb4d16b7ace2b1587
  Cargo.toml                    5bb6013a82e8b3a239d8efaed7e21d57c46862e7b93cf275a85f397e6b73bb05
  patch                         ba80a12846bc62a85d95878906aefd6c34614ffc25a0ea4a481b2187e34056e0
  src
    framework.rs -> ../../framework/framework.rs
    main.rs                     17c0c4e56e45645606e1a91f0d92d5c803fc6225e1638b9193211aefe963d376
os-jet-adv-02
  Cargo.lock                    0744c262252c23021fcb97451942268cf21c0b0baa5756feb4d16b7ace2b1587
  Cargo.toml                    5bb6013a82e8b3a239d8efaed7e21d57c46862e7b93cf275a85f397e6b73bb05
  patch                         ba80a12846bc62a85d95878906aefd6c34614ffc25a0ea4a481b2187e34056e0
  src
    framework.rs -> ../../framework/framework.rs
    main.rs                     af816b28514e3a09a31565660ea4319c1a42133888cde223709c791671c2718f
os-jet-adv-03
  Cargo.lock                    0744c262252c23021fcb97451942268cf21c0b0baa5756feb4d16b7ace2b1587
  Cargo.toml                    5bb6013a82e8b3a239d8efaed7e21d57c46862e7b93cf275a85f397e6b73bb05
  patch                         752ff461fa2fac581431505cf3d4184271490e05393b9531a01735a911bd15eb
  src
    framework.rs -> ../../framework/framework.rs
    main.rs                     33d9e21e2d8e1be15d0d88a75a870cb834b797d23f56d332ad34da06e8b171e0
os-jet-adv-04
  Cargo.lock                    0744c262252c23021fcb97451942268cf21c0b0baa5756feb4d16b7ace2b1587
  Cargo.toml                    5bb6013a82e8b3a239d8efaed7e21d57c46862e7b93cf275a85f397e6b73bb05
  patch                         752ff461fa2fac581431505cf3d4184271490e05393b9531a01735a911bd15eb
  src
    framework.rs -> ../../framework/framework.rs
    main.rs                     0a86113abad011d3880cc1ea4cfb57e065b3a9dc7b45939a72040dd59ef152b6
os-jet-adv-05
  Cargo.lock                    0744c262252c23021fcb97451942268cf21c0b0baa5756feb4d16b7ace2b1587
  Cargo.toml                    5bb6013a82e8b3a239d8efaed7e21d57c46862e7b93cf275a85f397e6b73bb05
  patch                         752ff461fa2fac581431505cf3d4184271490e05393b9531a01735a911bd15eb
  src
    framework.rs -> ../../framework/framework.rs
    main.rs                     2b1834d6d60f8338f0803ef6bc472335cc2590ff187eb2723e0493cba0323552
```

## Appendix D: OS-JET-ADV-01 Script

Proof of concept calculations for OS-JET-ADV-01.

```
var x = 100; // shares
var x2 = 100 + 100; // tokens

var old_tokens = x2;

var share_delta = 0;
var token_delta = 0;
var profit = 0;
for (let iter = 0; iter < 200; iter++) {
  console.log("state: " + x + " shares, " + x2 + " tokens");
  share_delta = 0;
  token_delta = 0;
  var max = 0;
  for (let y = 1; y < 1000; y++) {
    for (let z = 0; z < 1000; z++) {
      let tokens = y + z;
      let y_shares = Math.floor(y * x / x2);
      if (y_shares == 0) continue;

      while (y_shares >= Math.floor(tokens * (x + y_shares) / (x2 + y + z))) tokens++;

      if (tokens - y - z - 1 > max) {
        share_delta = y_shares;
        token_delta = y + z;
        console.log((y + z) + " tokens for -> " + tokens);
        max = Math.max(max, tokens - y - z - 1);
      }
    }
  }
  profit += max;
  x += share_delta;
  x2 += token_delta;
}
console.log("profit: ", profit);
console.log("cost: ", x2 - old_tokens);
```

## Appendix E: Vulnerability Rating Scale

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

| | |
|---|---|
| **Critical** | Vulnerabilities which **immediately** lead to loss of user funds with minimal preconditions<br><br>Examples:<br>- Misconfigured authority/token account validation<br>- Rounding errors on token transfers |
| **High** | Vulnerabilities which **could** lead to loss of user funds but are potentially difficult to exploit.<br><br>Examples:<br>- Loss of funds requiring specific victim interactions<br>- Exploitation involving high capital requirement with respect to payout |
| **Medium** | Vulnerabilities which could lead to denial of service scenarios or degraded usability.<br><br>Examples:<br>- Malicious input cause computation limit exhaustion<br>- Forced exceptions preventing normal use |
| **Low** | Low probability vulnerabilities which could still be exploitable but require extenuating circumstances or undue risk.<br><br>Examples:<br>- Oracle manipulation with large capital requirements and multiple transactions |
| **Informational** | Best practices to mitigate future security risks. These are classified as *general findings*.<br><br>Examples:<br>- Explicit assertion of critical internal invariants<br>- Improved input validation<br>- Uncaught Rust errors (vector out of bounds indexing) |