Phage
SECURITY

Bounce Tech

Security Review

Conducted by:

Pyro, Security Researcher
Samuraii77, Security Researcher
YanecaB, Junior Security Researcher

31.08.2025

Table of Contents

Disclaimer

Audits are a time, resource, and expertise-bound effort where trained experts evaluate smart contracts using a combination of automated and manual techniques to identify as many vulnerabilities as possible. Audits can reveal the presence of vulnerabilities **but cannot guarantee their absence**.

System overview

Bounce is a leveraged token protocol that creates synthetic leveraged exposure to perpetual futures markets. Users can mint/redeem leveraged tokens (like "ETH 10x Long") that automatically maintain target leverage ratios through integration with Hyperliquid's perpetual trading infrastructure, while earning fees that get distributed to BOUNCE token stakers.

Executive summary

Overview

Project Name	**Bounce Tech**
Repository	https://github.com/bounce-tech/bounce-contracts
Commit hash	b2761c71112ce13522e169ffc2bd9b0913d69e1c
Remediation	2bfeb6138e7081463a1210629df628ca8e4b5d30
Methods	Manual review

Timeline

Audit kick-off	18.08.2025
End of audit	22.08.2025
Remediations start	23.08.2025
Remediations end	31.08.2025

Scope

bounce-contracts/src/Airdrop.sol

bounce-contracts/src/Bounce.sol

bounce-contracts/src/Factory.sol

bounce-contracts/src/GlobalStorage.sol

bounce-contracts/src/HyperRead.sol

bounce-contracts/src/LeveragedToken.sol

bounce-contracts/src/LeveragedTokenProxy.sol

bounce-contracts/src/Locker.sol

bounce-contracts/src/Ownable.sol

bounce-contracts/src/Referrals.sol

bounce-contracts/src/Staker.sol

bounce-contracts/src/Vesting.sol

bounce-contracts/src/utils/ScaledNumber.sol

bounce-contracts/src/constants/Addresses.sol

bounce-contracts/src/constants/Config.sol

Issues Found

Severity	Count
High	4
Medium	7
Low	5

Findings

High Severity

[H-01] Spot and perp balance is not considered for calculating vault value

The total value of a vault (`LeveragedToken`) is computed by the idle assets in the contract and the value of our perps position on HyperCore.

The flow to get assets into a perps position is:

1. Transfer them out to our spot balance on HyperCore.
2. Transfer to our perps balance from our spot balance.
3. Use the perp balance to create perp positions.

The code does not include the funds in the spot balance which can be staying idle there waiting for a good trade entry. It also does not consider the perp balance, but only considers the perp position.

Resolution: Fixed in #PR52

[H-02] Stepwise jumps occur upon spot/perps balance top-ups, allowing MEV attacks

When funds are deposited into the `LeveragedToken` contract (which is basically a vault), a keeper will transfer the funds to our spot balance on HyperCore through the below function:

```
function withdrawBaseAsset(address to_, uint256 amount_) external override
   onlyKeepers {
   if (to_ == address(0)) revert InvalidAddress();
   if (amount_ == 0) revert InvalidAmount();
   if (amount_ > idleBaseAssetAmount()) revert InsufficientBalance();
   _checkpoint();
   _baseAsset().safeTransfer(to_, amount_);
   emit WithdrawBaseAsset(msg.sender, to_, amount_);
}
```

Then, these funds can be added to the perps balance on HyperCore or kept in the spot balance. However, due to the async design of HyperLiquid, funds disappear from our contract balance (as they are transferred out), but only appear in our spot balance in the next block. As this is not considered, the share value will unexpectedly go down each time `withdrawBaseAsset()` is called and will go back to normal when funds are in our perps position. This allows users or MEV bots to enter at the lower share value and exit at the higher one, resulting in profits for them and loss for the vault/other users.

Recommendation

Implement a credit tracker mapping, so when the keeper transfers out assets to spot balance, add that amount to a mapping with the current block number as the key. Then, adjust the total assets of the `LeveragedToken` contract to add that value.

Resolution: Fixed in #PR52

[H-03] Position value is computed incorrectly for shorts

Each perpetuals position has a size fetched and handles as follows:

```
int64 szi_ = position_.szi;
if (szi_ < 0) szi_ = -szi_;
uint256 unscaledSize_ = uint256(int256(szi_));
data_.size = unscaledSize_.scale(sizeDecimals_, 18);
```

A negative size represents a short position. When that is the case, the size is turned to a positive value. Then, the position value is computed as follows:

```
function positionValue(address user_, uint32 perp_) public view override
   returns (uint256) {
   PositionData memory data_ = positionData(user_, perp_);
   uint256 notional_ = data_.size.mul(data_.price);
   int256 pnl_ = int256(notional_) - int256(data_.initialNotional);
   int256 value_ = int256(data_.margin) + pnl_;
   if (value_ < 0) revert NegativePositionValue();
   return uint256(value_);
}
```

The PnL is computed by subtracting the initial notional from the current notional, where the notional is size times price. However, for a short, when the price has moved against our short (has went up), then the notional value will go up and thus increase the PnL - this is incorrect as price going up on a short should yield losses, not profits.

Resolution: Fixed in #PR33

[H-04] Unstaking operations are not validated against the creator

Unstakes happen in a 2-step process where it is first scheduled to be executed in the future (after a delay) and can then be executed in the future. However, when unstaking, a user can provide any ID:

```
function unstake(uint256 unstakeId_) external {
    _checkpoint(msg.sender);
    uint256 unstakeAmount_ = _unstake(msg.sender, unstakeId_, true);
    if (unstakeAmount_ == 0) revert NoPendingUnstake();
    _transferBounce(msg.sender, unstakeAmount_);
}
```

This allows the following to happen:

1. Bob unstakes his funds and must wait for the delay to pass.
2. The delay passes and Bob is about to complete the unstake.
3. Alice wants to unstake but does not want to wait for the delay. She creates an unstake with the amount of funds that correspond to Bob's unstake.
4. She immediately executes the unstake, but provides Bob's ID.
5. As Bob's ID is expired, the unstake will immediately go through and funds will go to Alice.
6. Bob can take Alice's unstake after it expires due to the same bug, however another user can execute the same attack on Alice's unstake and leave Bob in a state where he is indefinitely disallowed from receiving his funds.

Recommendation

To fix the issue, consider validating the creator of the stake against the one executing it. However, also note that the current mechanism for tracking unstakes is significantly less trivial than it should be for what it is, there are currently too many variables to track - consider simplifying it, for example instead of having separate structures for unstake positions, unstake data and the actual unstakes, consider making a simple mapping with an ID as a key and data as the value and then allow users to provide that ID (with validations for the creator of course).

Resolution: Fixed in #PR38

Medium Severity

[M-01] DoS in `prepareRedeem` due to fee exceeding contract assets

The `prepareRedeem` function is designed for cases where the contract does not hold enough assets to pay out immediately, allowing users to queue their redemption for later execution. However, the `_redemptionFee` is calculated as `baseAmount * redemptionFee * targetLeverage`. Since the contract only keeps around 10% of the assets in its balance, large redemptions can cause the computed fee to exceed the contract's available balance.

In such cases, `_payRedemptionFee` will revert, making it impossible for users to even prepare a redemption because the fee is paid before the tokens are transferred to the contract. This results in a DoS scenario that defeats the purpose of `prepareRedeem`.

```
function _redemptionFee(uint256 baseAmount_) internal view returns (
   uint256) {
   if (baseAmount_ == 0) return 0;
   LeveragedTokenStorage storage $ = _getLeveragedTokenStorage();
   return baseAmount_.mul($.globalStorage.redemptionFee()).mul(
      $.targetLeverage);
}
```

Recommendation

Consider creating a new variant of `_payFeesAndBurn()` (used specifically when preparing a redeem) which adds the pre-fee amount as credit without transferring out any fees in the preparation stage and then process the fee upon the actual redemption execution.

Resolution: Fixed in #PR43

[M-02] Rewards are lost due to rounding in `donateFees`

When users redeem, they pay the redemption fees with `_payRedemptionFee`, where part of the fees are sent to referrals and the other part to the staking contract.

However, in `donateFees` the math we implement will result in rounding when amounts are small, which is even further amplified by the fact that our fees are in base token (USDC) which uses 6 decimals for precision. Inside `donateFees` we can see how we calculate `_integral -> amount (in 6) * 1e18 / (totalStaked - totalUnstaking) (in 1e18)`

```
function donateFees(uint256 amount_) external override {
   if (amount_ == 0) return;
   uint256 div_ = totalStaked - totalUnstaking;
   if (div_ == 0) revert ZeroBalance();
   baseAsset().safeTransferFrom(msg.sender, address(this), amount_);
   _integral += amount_.div(div_);
   emit DonateStakeReward(msg.sender, amount_, div_, _integral);
}
```

Example:

1. Redemption fee is 1%
2. User withdraws 5 USDC - 5e6 (the min deposit amount)
3. There are 51k staked bounce inside `Staker.sol`
4. The math will calculate an increase in integral by `0`

```
5e6 * 1% = 5e4 - fee
5e4 * 1e18 / 51000e18 = 0.98 => 0
```

5. The fee is still sent to the contract, it's just not reflected as such

Note that the same issue occurs inside `Locker` too, but it's to a lower degree since the scale which we use is 18.

Recommendation

Consider keeping the leftover and saving it for the next time.

```
    uint256 internal _integral;
    uint256 internal _unstakeIndex;
+   uint256 internal _accumulatedDust; // Accumulated precision loss from
   rounding
    // ...
    function donateFees(uint256 amount_) external override {
       if (amount_ == 0) return;
       uint256 div_ = totalStaked - totalUnstaking;

       if (div_ == 0) revert ZeroBalance();
       baseAsset().safeTransferFrom(msg.sender, address(this), amount_);
-      _integral += amount_.div(div_);
+      // Add any accumulated dust from previous rounds
+      uint256 totalAmount_ = amount_ + _accumulatedDust;
+      // Calculate integral increase and track leftover
+      uint256 integralIncrease_ = totalAmount_.div(div_);
+      uint256 distributed_ = integralIncrease_.mul(div_);
+      // Store leftover for next round
+      _accumulatedDust = totalAmount_ - distributed_;
+      _integral += integralIncrease_;

       emit DonateStakeReward(msg.sender, amount_, div_, _integral);
    }
```

Resolution: Fixed in #PR27

[M-03] First depositor in the locker can claim past rewards

In the `Locker` contract, funds are distributed each second to the depositors. The distribution starts through `startDistribution()` at the current block timestamp. This means that each lock (unless in the very same block as the distribution start) will be done after some rewards have already been distributed.

The issue is that the first depositor unfairly gets the already distributed rewards before his deposit, because of how the logic behaves. He can do the following:

1. Distribution has started, wait until the first user tries to lock which can be let's say a day later.
2. When the first user tries to lock, frontrun him by doing the following:

- lock 1 wei.
- claim the rewards.

When that is done, the malicious user will claim all of the already accrued rewards as during the first deposit, the integral and accounted rewards are not updated.

PoC:

```
function testProfits() public {
   locker.startDistribution();

   skip(1 days); // 1 day after distribution has started.

   address user = makeAddr('user');
   _mintTokens(address(bounce), user, 1);

   vm.startPrank(user);
   bounce.approve(address(locker), 1);
   locker.lock(1);
   uint256 balBefore = bounce.balanceOf(user);
   locker.claim();
   uint256 balAfter = bounce.balanceOf(user);

   uint256 received = balAfter - balBefore;
   assertEq(219780219780219760000000, received);
}
```

Recommendation

The simplest fix without changing any actual logic is to implement a minimum lock amount. This will work good because the first user will be incentivized to deposit to get these initial rewards for himself, however it will not be in a malicious manner where he simply locks 1 wei just to steal the rewards.

Resolution: Fixed in #PR28

[M-04] Revocation can be bypassed by front-running with `Vesting::transfer`

In the `Vesting` contract, the owner can revoke a user's vesting by calling `revoke()` to prevent them from increasing their claimable tokens. However, a user can front-run the owner's transaction and call `transfer()` to move their vesting to another address not yet listed in the `_claimed` mapping. By doing this, the user escapes revocation and continues accruing claimable tokens, which they can later claim using the new address.

```
function transfer(address to_) external override {
      address from_ = msg.sender;
      if (to_ == address(0)) revert InvalidAddress();
      if (from_ == to_) revert InvalidAddress();
      VestingConstants memory fromConstants_ = _data[from_];
      if (fromConstants_.revokedAt != 0) revert VestingRevoked();
      if (fromConstants_.start == 0) revert NoVesting();
      VestingConstants memory toConstants_ = _data[to_];
      if (toConstants_.start != 0) revert VestingAlreadyExists();

      _data[to_] = VestingConstants({start: fromConstants_.start,
         amount: fromConstants_.amount, revokedAt: 0});
      delete _data[from_];
      _claimed[to_] = _claimed[from_];
      delete _claimed[from_];
      emit TransferVesting(from_, to_, fromConstants_.amount);
   }
```

Recommendation

Implement a `prepareTransfer` function that introduces a delay before the actual transfer, similar to `Staker::prepareUnstake`.

Resolution: Fixed in #PR29

[M-05] `prepareRedeem` does not work as intended

`prepareRedeem` is used to schedule redeems when there is insufficient free balance (`_baseAsset().balanceOf(address(this)) - debt`) inside the contract.

However, this `if` inside its internal function (`_payFeesAndBurn`) prevents us from scheduling more assets than there are currently free inside the contract.

```
      if (baseAmount_ > idleBaseAssetAmount()) revert InsufficientBalance();
```

This makes this function operate under the same conditions as `redeem`, and thus users are unable to request their assets for redemption if the contract does not have these assets.

Recommendation

The fix for this would require `_payFeesAndBurn` to have a way to differentiate from which function it's called (i.e., a variable can be added), where if called from `prepareRedeem` it would skip this check:

```
if (baseAmount_ > idleBaseAssetAmount()) revert InsufficientBalance();
```

However, now the debt can exceed the contract balance if users schedule enough at the same time. This means that `idleBaseAssetAmount` must be able to handle negative values, as well as the function that uses it - `totalValue`.

Resolution: Fixed in #PR30

[M-06] Fees can grow over total idle assets, causing DoS

Fees are calculated as follows in `LeveragedToken`:

```
uint256 annualFee_ = totalAssets().mul(streamingFee_).mul($.targetLeverage);
uint256 periodFee_ = annualFee_.mul(percentOfYear_);
```

The total assets are all of the vaults owned by the vault, including idle balance and perp position. The issue is that the fees can currently grow over the total assets if enough time passes. The likelihood of this happening is not small as the total assets includes funds in the perp position and the idle balance, while fees are transferred only from the idle balance, so if there is a 10% pending fee, but only 9% of the total assets are in the vault, then the code would revert as there are insufficient assets to cover the fees.

Recommendation

To fix the issue, consider capping the fee to be a % of the idle assets, i.e. 100% of the idle assets (so just cap at idle assets) or to be more favorable to users, let's say 50% of the idle assets.

Resolution: Fixed in #PR26

[M-07] Removed referrers can continue receiving rebates from already-joined users

Removing a referrer via `Referrals::removeReferrer` deletes their referral code but does not affect existing users who joined with it. These users' `_referreeReferrers` still point to the removed referrer, allowing them to continue receiving and claiming rebates through `donateRebates` and `claimRebates`. This occurs because `donateRebates` does not check whether the `referrer_` returned from `_referreeReferrers[user_]` is still valid, allowing removed referrers to keep receiving donations.

Recommendation

Add a check in `donateRebates` to ensure `_referreeReferrers[user_]` points to an active referrer.

```
function donateRebates(address user_, uint256 feeAmount_) external override
   returns (uint256) {
      address referrer_ = _referreeReferrers[user_];
-     if (referrer_ == address(0)) return 0;
+     if (referrer_ == address(0) || bytes(_referrerCodes[referrer_]).length
   == 0) return 0;
}
```

And allow users to rejoin with a new referral code only if their current referrer has been removed.

```
   function joinWithReferral(string calldata referralCode_) external override {
      if (bytes(referralCode_).length == 0) revert InvalidReferralCode();
      bool hasJoined_ = _referreeReferrers[msg.sender] != address(0);
-     if (hasJoined_) revert UserAlreadyJoined();
+     if (hasJoined_ && bytes(_referrerCodes[_referreeReferrers[msg.sender]]).
   length > 0) revert UserAlreadyJoined();
}
```

Resolution: Fixed in #PR34

Low Severity

[L-01] Minting rounds in favor of the user

The following function is used to turn assets into shares when minting:

```
function baseToLtAmount(uint256 baseAmount_) public view override returns (
   uint256) {
   uint256 scaled_ = baseAmount_.scaleFrom(_baseAsset().decimals());
   return scaled_.div(exchangeRate());
}
```

The exchange rate is as follows:

```
return totalValue().div(totalSupply());
```

As the exchange rate uses division, it can round down. Then, as the exchange rate is used as a denominator in `baseToLtAmount()` and it is potentially rounded down, it will actually round up the equation there, resulting in the system potentially rounding in favor of the user.

Recommendation

Under normal circumstances, this should not pose issues and a fix, while recommended, is not completely necessary. If you wish to fix the issue, consider refactoring `scaled_.div(exchangeRate())` to subtract 1 if the division is not perfect.

Resolution: Acknowledged

[L-02] Minting `LeveragedToken` is DoSed when there are no stakers

When a user tries to mint in `LeveragedToken`, a streaming fee is paid to the `Staker` contract:

```
function _payFees(uint256 baseAmount_) internal {
   if (baseAmount_ == 0) return;
   LeveragedTokenStorage storage $ = _getLeveragedTokenStorage();
   _baseAsset().safeIncreaseAllowance(address($.globalStorage.staker()),
      baseAmount_);
   $.globalStorage.staker().donateFees(baseAmount_);
}
```

However, when `donateFees()` is called on the staker contract, we revert if there are no active stakes:

```
uint256 div_ = totalStaked - totalUnstaking;
if (div_ == 0) revert ZeroBalance();
```

Recommendation

As depending on the fix, new issues could be introduced, such as stakers receiving fees for times they weren't staked in, we recommend acknowledging the issue as the likelihood of there not being a staker is low and if it happened, a trusted entity, like the owner, can stake some amount himself.

Resolution: Acknowledged

[L-03] `minSize` check is not fully effective

Upon minting and redeeming, the following check can be seen:

```
if (baseAmount_ < $.globalStorage.minSize()) revert BelowMinSize();
```

The user is unable to deposit amount less than `minSize`. However, he can still:

1. Deposit `minSize + X`.
2. Redeem shares equal to `minSize` amount.
3. End result is `X` amount of assets deposited, where `X` can be an amount less than `minSize`, effectively bypassing the check.

Recommendation

For the fix of the issue, one is not necessarily needed and the issue can be acknowledged as there is no significant impact (except for the vault inflation issue which is a separate one).

Resolution: Acknowledged

[L-04] Inflation attack is possible in `LeveragedToken`

The `LeveragedToken` contract works like a vault and it is vulnerable to a typical vault inflation attack, causing a loss for the first depositor:

1. Victim deposits 100e6 tokens.
2. Attacker frontruns, deposits `1e6 + 1` amount of assets (we assume `minSize` is 1e6). This mints him `1000001000000000000` shares. This is to bypass the `minSize` check.
3. Attacker redeems `1000000999999999999` shares which gives him 1e6 assets. State is 1 share and 1 asset now.
4. Attacker directly transfers 100e6 tokens to the contract, now assets are `100e6 + 1`.
5. The victim's deposit results in 0 shares minted as `100e18 * 1e18 / (((100e6 + 1) * 1e12) * 1e18 / 1) = 0`
6. Attacker steals his funds by redeeming.

The issue is marked as a Low as there is a slippage protection upon minting, however the issue can still technically happen if the depositor does not utilize the slippage functionality and provides 0 as the minimum shares out.

Recommendation

To fix the issue, consider minting some shares as the first depositor upon deployment. Another issue is the fact that `minSize` check can be bypassed by depositing and immediately redeeming.

Resolution: Fixed in #PR48

[L-05] Vesting creation DoS by transferring expired vests

Creating a vest for a user requires that one does not already exist:

```
if (constants_.start ≠ 0) revert VestingAlreadyExists();
```

However, users are able to transfer their vests to other users using `Vesting::transfer()` in order to DoS vest creation for the receiver. In the usual case, this would cause a loss of funds for them as they are losing out on the unvested and unclaimed part of their vest, however there is nothing stopping them from transferring out their claimed and expired vests, resulting in no loss of funds for them.

Recommendation

To fix the issue, consider implementing a function that allows users to delete their expired vests.

Resolution: Fixed in #PR29