Pashov Audit Group

Legion
Security Review

July 28th 2025 - August 6th 2025

Contents

1. About Pashov Audit Group 3
2. Disclaimer 3
3. Risk Classification 3
4. About Legion 4
5. Executive Summary 4
7. Findings 5

High findings 7

[H-01] User may transfer position and withdraw project capital 7
[H-02] `LegionLinearEpochVesting` breaks vesting schedule 8
[H-03] Excess epoch progression might lock funds permanently 11
[H-04] Malicious delays possible in epoch vesting schedule 13

Medium findings 16

[M-01] Encrypted amountOut in `invest()` can be guessed by brute-force 16
[M-02] Transferring to refunded recipient corrupts cache and locks funds 17
[M-03] Transferred investment position is not claimable 19

Low findings 21

[L-01] `LegionPositionManager` violates `ERC-721` by ignoring invalid tokens 21
[L-02] `invest()` missing `_verifyCanClaimExcessCapital()` enables reinvestment 21
[L-03] `LegionBouncer` update relies on outdated bouncer post `ID` change 21
[L-04] `emergencyTransferOwnership()` can be front-run to inaccessible wallets 22
[L-05] Changing `vestingController` not retroactive for contracts 22
[L-06] No expiration or invalidation controls for investment signatures 23
[L-07] Paused `releaseVestedTokens()` can be bypassed 23

1. About Pashov Audit Group

Pashov Audit Group consists of 40+ freelance security researchers, who are well proven in the space - most have earned over $100k in public contest rewards, are multi-time champions or have truly excelled in audits with us. We only work with proven and motivated talent.

With over 300 security audits completed — uncovering and helping patch thousands of vulnerabilities — the group strives to create the absolute very best audit journey possible. While 100% security is never possible to guarantee, we do guarantee you our team's best efforts for your project.

Check out our previous work [here](#) or reach out on Twitter [@pashovkrum](#).

2. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

3. Risk Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

Impact

- **High** - leads to a significant material loss of assets in the protocol or significantly harms a group of users
- **Medium** - leads to a moderate material loss of assets in the protocol or moderately harms a group of users
- **Low** - leads to a minor material loss of assets in the protocol or harms a small group of users

Likelihood

- **High** - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost
- **Medium** - only a conditionally incentivized attack vector, but still relatively likely
- **Low** - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive

4. About Legion

Legion is a fundraising platform that deploys sale and vesting contracts supporting Fixed Price, Sealed Bid, and Pre-Liquid offerings. It enforces investor eligibility via Merkle Proofs and signatures, while interacting with off-chain backend calculations to coordinate merit-based capital raising before, during, and after token generation events.

5. Executive Summary

A time-boxed security review of the **Legion-Team/legion-protocol-contracts** repository was done by Pashov Audit Group, during which Pashov Audit Group engaged to review **Legion**. A total of **14** issues were uncovered.

Protocol Summary	
Project Name	Legion
Protocol Type	Fundraising platform
Timeline	July 28th 2025 - August 6th 2025

Review commit hash:

- 30a2b29fb9ad6a53eb9731be6ae3e0255871315e (Legion-Team/legion-protocol-contracts)

Fixes review commit hash:

- 85d479ea08d148a380138b535ed11768adee16de (Legion-Team/legion-protocol-contracts)

Scope

`LegionTokenDistributor.sol` `LegionReferrerFeeDistributor.sol` `LegionPositionManager.sol` `LegionCapitalRaise.sol` `LegionAbstractSale.sol` `LegionFixedPriceSale.sol` `LegionPreLiquidApprovedSale.sol` `LegionPreLiquidOpenApplicationSale.sol` `LegionSealedBidAuctionSale.sol` `LegionLinearEpochVesting.sol` `LegionLinearVesting.sol` `LegionVestingManager.sol` `ERC5192.sol`

6. Findings

Findings count

Severity	Amount
High	4
Medium	3
Low	7
Total findings	**14**

Summary of findings

ID	Title	Severity	Status
[H-01]	User may transfer position and withdraw project capital	High	Resolved
[H-02]	`LegionLinearEpochVesting` breaks vesting schedule	High	Resolved
[H-03]	Excess epoch progression might lock funds permanently	High	Resolved
[H-04]	Malicious delays possible in epoch vesting schedule	High	Resolved
[M-01]	Encrypted amountOut in `invest()` can be guessed by brute-force	Medium	Resolved
[M-02]	Transferring to refunded recipient corrupts cache and locks funds	Medium	Resolved
[M-03]	Transferred investment position is not claimable	Medium	Resolved
[L-01]	`LegionPositionManager` violates `ERC-721` by ignoring invalid tokens	Low	Resolved
[L-02]	`invest()` missing `_verifyCanClaimExcessCapital()` enables reinvestment	Low	Resolved
[L-03]	`LegionBouncer` update relies on outdated bouncer post `ID` change	Low	Acknowledged
[L-04]	`emergencyTransferOwnership()` can be front-run to inaccessible wallets	Low	Acknowledged

ID	Title	Severity	Status
[L-05]	Changing `vestingController` not retroactive for contracts	Low	Acknowledged
[L-06]	No expiration or invalidation controls for investment signatures	Low	Acknowledged
[L-07]	Paused `releaseVestedTokens()` can be bypassed	Low	Acknowledged

High findings

[H-01] User may transfer position and withdraw project capital

Severity

Impact: High

Likelihood: Medium

Description

After the sale results are published, each investor is assigned: 1. An **accepted capital** (used to buy tokens). 2. An **allocated token amount.**

Investors should: - Call `claimTokenAllocation()` to claim tokens (while leaving accepted capital in the contract). - Call `withdrawExcessInvestedCapital()` to withdraw the **excess** amount (**invested - accepted**).

However, there is a flaw when using `transferInvestorPositionWithAuthorization()` :

If a user transfers their investment position after the result is published, the new receiver's position gets their own `investedCapital` increased with the transferred amount. But their accepted capital remains unchanged, because it's tied to the original investor's allocation.

This mismatch enables the receiver to call `withdrawExcessInvestedCapital()` and withdraw all of the newly added `investedCapital` as if it were excess, even though part of it was meant to be **accepted capital** reserved for token allocation. As a result, funds that should remain locked for the project can be withdrawn and drained.

```
function withdrawExcessInvestedCapital(
    uint256 amount,
    bytes32[] calldata proof
)
    external
    virtual
    whenNotPaused
    whenSaleNotCanceled
{
    --Snipped--

    // Verify that the investor is eligible to get excess capital back
    _verifyCanClaimExcessCapital(msg.sender, positionId, amount, proof);
    --Snipped--
}
```

```
function _verifyCanClaimExcessCapital(
    address _investor,
    uint256 _positionId,
    uint256 _amount,
```

```
        bytes32[] calldata _proof
    )
        internal
        view
        virtual
    {
        // Load the investor position
        InvestorPosition memory position = s_investorPositions[_positionId];

        // Check if the investor has already settled their allocation
        if (position.hasClaimedExcess) revert
Errors.LegionSale__AlreadyClaimedExcess(_investor);

        // Generate the merkle leaf and verify accepted capital
@>      bytes32 leaf = keccak256(bytes.concat(keccak256(abi.encode(_investor,
(position.investedCapital - _amount))))); //@audit: only checks the final amount (accepted
capital) - does not check the amount being withdrawn

        // Verify the merkle proof
        if (!MerkleProofLib.verify(_proof, s_saleStatus.acceptedCapitalMerkleRoot, leaf)) {
            revert Errors.LegionSale__CannotWithdrawExcessInvestedCapital(_investor, _amount);
        }
    }
```

Recommendations

Choose one of these solutions to fix the issue:

1- Independent Merkle Proof Validation Per Position Positions should be transferable, but each position must have its own Merkle proof to verify accepted capital and related data independently. The ownership of the position should be separate from the position data itself, allowing a single user to hold multiple positions without conflicts. This ensures every position's state remains valid regardless of transfers, and the user can claim and verify all of their positions separately.

2- Two-Step Merkle Proof Alternatively, implement a two-step Merkle proof process. First, set a Merkle proof before any transfers occur, which allows position transfers but disables claiming excess capital. After transfers are finalized and locked, set a second Merkle proof to enable secure excess capital claims. This approach lets users trade positions while preventing abuse of excess capital withdrawals.

[H-02] `LegionLinearEpochVesting` breaks vesting schedule

Severity High

Impact: Medium

Likelihood: High

Description

The `_vestingSchedule` override in `LegionLinearEpochVesting` does not comply with the expected behavior defined by the base contract `VestingWalletUpgradeable`. As a result, it breaks the core invariant of the `release()` logic and may lead to multiple math errors or underflows when claiming vested tokens.

In `VestingWalletUpgradeable`, the `_vestingSchedule()` function is meant to return the total vested amount (i.e., claimed + unclaimed) as of a given timestamp. The `release()` function uses it to compute how much can be claimed by subtracting the amount already released.

`VestingWalletUpgradeable::_vestingSchedule` In the snippet provided below, we can see that the time delta used in the calculation is with respect to the start time. Hence, we should expect a cumulative return from startime

```
    /**
     * @dev Virtual implementation of the vesting formula. This returns the amount vested, as a
function of time, for
     * an asset given its total historical allocation.
     */
    function _vestingSchedule(uint256 totalAllocation, uint64 timestamp) internal view virtual
returns (uint256) {
        if (timestamp < start()) {
            return 0;
        } else if (timestamp >= end()) {
            return totalAllocation;
        } else {
@>>         return (totalAllocation * (timestamp - start())) / duration();
        }
    }
```

Instead of returning a cumulative total, the override only returns the amount newly vested since the last claimed epoch (`s_lastClaimedEpoch`). It ignores previously released amounts and violates the expectations of the `vestedAmount()` function.

Calculation Example

Given values: - totalAllocation = 10000 - s_numberOfEpochs = 5 - currentEpoch = 2 (getCurrentEpochAtTimestamp returns 2 after epoch 1) - s_lastClaimedEpoch = 0

Calculation for Epoch 2

amountVested = (2 - 1 - 0) * (10000 / 5)
amountVested = 2000

After this step:
released = 2000

Claiming in the next epoch

amountVested = (3 - 1 - 1) * (10000 / 5)
amountVested = 2000

releasable = amountVested - released
releasable = 2000 - 2000
releasable = 0

As we can see in the above example, in the next epoch, releasable yields 0. Hence we have no tokens to claim. This will repeat over epochs continuously, until the last vesting epoch when the `_vestingSchedule` returns `_totalAllocation`.

```
    function _vestingSchedule(
        uint256 _totalAllocation,
        uint64 _timestamp
    )
        internal
        view
        override
        returns (uint256 amountVested)
    {
        // Get the current epoch
        uint256 currentEpoch = getCurrentEpochAtTimestamp(_timestamp);

        // If all epochs have elapsed, return the total allocation
        if (currentEpoch >= s_numberOfEpochs + 1) {
            amountVested = _totalAllocation;
        }

        // Otherwise, calculate the amount vested based on the current epoch
        //@audit bug this is most likely incorrect
        if (currentEpoch > s_lastClaimedEpoch) {
            //@audit it does not integrate well into the calculation of vested amount
            amountVested = ((currentEpoch - 1 - s_lastClaimedEpoch) * _totalAllocation) /
s_numberOfEpochs;
        }
    }
```

```
    function release() public virtual {
        VestingWalletStorage storage $ = _getVestingWalletStorage();
@>>     uint256 amount = releasable();
        $._released += amount;
        emit EtherReleased(amount);
        Address.sendValue(payable(owner()), amount);
    }
```

```
    function releasable() public view virtual returns (uint256) {
        //@audit an underflow is likely to happen here since released can be greater than the
value returned from `vestedAmount`
@>>     return vestedAmount(uint64(block.timestamp)) - released();
    }
```

```
    function vestedAmount(uint64 timestamp) public view virtual returns (uint256) {
        return _vestingSchedule(address(this).balance + released(), timestamp);
    }
```

NOTE : The impact of this finding is worse if the user cliffs for a few epochs before the start. After the first distribution (after cliff ends), the user may not be able to release tokens until later epochs. i.e, if the user cliffs for n epochs, claim can only occur after (2n + 1) epochs

Recommendations

Replace this line in `_vestingSchedule`.

```
-             amountVested = ((currentEpoch - 1 - s_lastClaimedEpoch) * _totalAllocation) /
s_numberOfEpochs
+            amountVested = ((currentEpoch - 1 ) * _totalAllocation) / s_numberOfEpochs
```

[H-03] Excess epoch progression might lock funds permanently

Severity

Impact: High

Likelihood: Medium

Description

The Linear epoch vesting contracts allow users to receive tokens as per the epochs mentioned in the investor terms via `LegionLinearEpochVesting::release()`.

However, the `_vestingSchedule()` has a logical flaw which would overestimate the `amountVested` if the `currentEpoch` is greater than the `s_numberOfEpochs`:

```
    function _vestingSchedule(
        uint256 _totalAllocation,
        uint64 _timestamp
    )
        internal
        view
        override
        returns (uint256 amountVested)
    {
        // Get the current epoch
        uint256 currentEpoch =
getCurrentEpochAtTimestamp(_timestamp);                             <<@

        // If all epochs have elapsed, return the total allocation
        if (currentEpoch >= s_numberOfEpochs + 1) {
            amountVested = _totalAllocation;                         <<@ - // Fails to return
        }

        // Otherwise, calculate the amount vested based on the current epoch
        if (currentEpoch > s_lastClaimedEpoch) {
            amountVested = ((currentEpoch - 1 - s_lastClaimedEpoch) * _totalAllocation) /
```

```
s_numberOfEpochs;                <<@
        }
    }
```

The control flow fails to return whenever the `currentEpoch` is greater than `s_numberOfEpochs`, hence, the next if statement gets triggered, leading to `(currentEpoch - 1 - s_lastClaimedEpoch) > s_numberOfEpochs`, eventually over-reporting `amountVested`.

Proof of Concept (PoC):

1. A user gets their token vested via the `LegionLinearEpochVesting` contract in a normal sales contract flow.
2. The user fails to claim tokens before the last epoch ends.
3. Calling the `release()` function would revert due to insufficient funds.

```
    function test_broken_release_after_extra_epochs() public {
        // Arrange
        prepareCreateLegionLinearEpochVesting();

        // Extra 1 epoch to break the release logic
        vm.warp(block.timestamp + 2_678_400 * 13 + 1);

        // This would work as expected
        // vm.warp(block.timestamp + 2_678_400 * 12 + 1);

        // Expect
        // vm.expectEmit();
        // emit VestingWalletUpgradeable.ERC20Released(address(askToken), 1200 * 1e18);

        // Reverts with `InsufficientBalance()`
        vm.expectRevert();
        LegionLinearEpochVesting(payable(legionVestingInstance)).release(address(askToken));
    }
```

Recommendations

It is recommended to return if the first if condition `currentEpoch >= s_numberOfEpochs + 1` is satisfied:

```
    function _vestingSchedule(
        uint256 _totalAllocation,
        uint64 _timestamp
    )
        internal
        view
        override
        returns (uint256 amountVested)
    {
        // Get the current epoch
        uint256 currentEpoch = getCurrentEpochAtTimestamp(_timestamp);

        // If all epochs have elapsed, return the total allocation
```

```
        if (currentEpoch >= s_numberOfEpochs + 1) {
            amountVested = _totalAllocation;
+           return;
        }
        . . .
    }
```

[H-04] Malicious delays possible in epoch vesting schedule

Severity

Impact: High

Likelihood: Medium

Description

The `LegionLinearEpochVesting::release()` allows users to directly release particular tokens as per the designated vesting schedule:

```
    function release(address token) public override onlyCliffEnded {
        super.release(token);

        // Update the last claimed epoch
        _updateLastClaimedEpoch();
    }
```

The `s_lastClaimedEpoch` plays a crucial role as the release is dependent upon comparing `currentEpoch` :

```
    function _vestingSchedule(
        uint256 _totalAllocation,
        uint64 _timestamp
    )
        internal
        view
        override
        returns (uint256 amountVested)
    {
        // Get the current epoch
        uint256 currentEpoch = getCurrentEpochAtTimestamp(_timestamp);

        // If all epochs have elapsed, return the total allocation
        if (currentEpoch >= s_numberOfEpochs + 1) {
            amountVested = _totalAllocation;
        }

        // Otherwise, calculate the amount vested based on the current epoch
        if (currentEpoch > s_lastClaimedEpoch)
{                                                          <<@
            amountVested = ((currentEpoch - 1 - s_lastClaimedEpoch) * _totalAllocation) /
```

```
s_numberOfEpochs;          <<@
        }
    }
```

Hence, an attacker can manipulate the `s_lastClaimedEpoch` by calling `release()` with any other token, which would trigger the `_updateLastClaimedEpoch()` and update the `s_lastClaimedEpoch`, denying the original token to be released.

```
    function _updateLastClaimedEpoch() internal {
        // Get the current epoch
        uint64 currentEpoch = getCurrentEpoch();

        // If all epochs have elapsed, set the last claimed epoch to the total number of epochs
        if (currentEpoch >= s_numberOfEpochs + 1) {
            s_lastClaimedEpoch = s_numberOfEpochs;
            return;
        }

        // If the current epoch is greater than the last claimed epoch, set the last claimed
epoch to the current epoch - 1
        s_lastClaimedEpoch = currentEpoch -
1;
<<@
    }
```

This would allow the attacker to artificially keep the market token price inflated; hence, they have a decent motive to perform the attack.

Proof of Concept (PoC):

1. The legitimate user's epoch is about to end.
2. Attacker frontruns or simply calls `release()` function with a different token address than intended.
3. This would move the `s_lastClaimedEpoch` forward, denying the intended token's release.

Add the following test case inside the `LegionLinearEpochVesting.t.sol` file:

```
    function test_delay_grief_attack() public {
        // Arrange
        prepareCreateLegionLinearEpochVesting();

        vm.warp(block.timestamp + 2_678_400 + 1);

        // Balance before release
        uint balanceBefore = MockERC20(askToken).balanceOf(vestingOwner);

        // Malicious actor delays the release by calling it with a different token address
        MockERC20 maliciousToken = new MockERC20("Malicious Token", "MAL", 18);
        vm.prank(address(0x05));

LegionLinearEpochVesting(payable(legionVestingInstance)).release(address(maliciousToken));

        // Legitimate user releases the tokens
        vm.prank(vestingOwner);
```

```
        LegionLinearEpochVesting(payable(legionVestingInstance)).release(address(askToken));

        uint balanceAfter = MockERC20(askToken).balanceOf(vestingOwner);

        // Balance before and after release should be the same
        assertEq(balanceBefore, balanceAfter);

    }
```

Recommendations

It is recommended to set the ask token while deploying the vesting contract and only allow calling that particular token param in the `release()` call, or only allow the owner of the vesting contract to call the `raise()` by implementing an `onlyOwner` modifier.

Medium findings

[M-01] Encrypted amountOut in `invest()` can be guessed by brute-force

Severity

Impact: High

Likelihood: Low

Description

In the `invest()` function of `LegionSealedBidAuctionSale`, users submit their investment along with a `sealedBid`, which includes:

- An encrypted amount (`encryptedAmountOut`).
- A random number (`salt`).
- A public key (`sealedBidPublicKey`).

This is supposed to hide the real rate the user wants to invest (like in a sealed-bid auction). But the problem is:

- The encrypted amount, salt, and public key are decoded and exposed during the `invest()` call.
- They are also **emitted in an event** (`CapitalInvested`), which anyone can see.

Because of this, someone could:

1. Watch the blockchain or mempool for incoming bids.
2. Use the known values to brute-force off-chain.
3. Eventually figure out how much someone really invested.

This breaks the idea of a private, sealed bid and could let attackers cheat in the auction.

Recommendations

Use stronger encryption so attackers can't guess the real amount. One solution can be to allow the user to encrypt a pair value `(bid amount, random number)`, and this way the user can choose different random numbers, and the result of encryption would be different for the same bid amounts, and an attacker can't brute force it.

[M-02] Transferring to refunded recipient corrupts cache and locks funds

Severity

Impact: High

Likelihood: low

Description

The `transferInvestorPosition()` and `transferInvestorPositionWithAuthorization()` allow legion bouncer and users to transfer their positions to others for the purpose of OTC sell-offs. However, while transferring the position, the recipient is not checked to see if there was any kind of refund made in the past. This is important due to the fact that `refund()` calls across sales and raise contracts do not set the cached amount to 0.

Hence, the `_burnOrTransferInvestorPosition` call would increase `cachedTokenAllocationRate` and `cachedInvestAmount` falsely.

```
    function _burnOrTransferInvestorPosition(address _from, address _to, uint256 _positionId)
private {
        // Get the position ID of the receiver
        uint256 positionIdTo = s_investorPositionIds[_to];

        // If the receiver already has a position, burn the transferred position
        // and update the existing position
        if (positionIdTo != 0) {
            // Load the investor positions
            InvestorPosition memory positionToBurn = s_investorPositions[_positionId];
            InvestorPosition storage positionToUpdate = s_investorPositions[positionIdTo];

            // Update the existing position with the transferred values
            positionToUpdate.investedCapital += positionToBurn.investedCapital;
            positionToUpdate.cachedTokenAllocationRate +=
positionToBurn.cachedTokenAllocationRate;               <<@
            positionToUpdate.cachedInvestAmount +=
positionToBurn.cachedInvestAmount;                              <<@

            // Delete the burned position
            delete s_investorPositions[_positionId];

            // Burn the investor position from the `from` address
            _burnInvestorPosition(_from);
        } else {
            // Transfer the investor position to the new address
            _transferInvestorPosition(_from, _to, _positionId);
        }
    }
```

This would impact the protocol in two ways:

1. The SAFT requirement to keep cached amounts in sync with the investment amount would be broken.

```
investedCapital != cachedInvestAmount
```

1. The `LegionPreLiquidApprovedSale::claimTokenAllocation` requires passing on actual values of the `investAmount` and `tokenAllocationRate` instead of what's being stored as cache to avoid failure, hence, requires an off-chain mechanism to keep track in such cases separately.
2. If a sale is cancelled post such OTC sales, `withdrawInvestedCapitalIfCanceled()` would fail due to `_verifyHasNotRefunded()` check, leading to stuck funds:

```
    function withdrawInvestedCapitalIfCanceled() external whenNotPaused whenSaleCanceled {
        // Get the investor position ID
        uint256 positionId = _getInvestorPositionId(msg.sender);

        // Verify that the position exists
        _verifyPositionExists(positionId);

        // Verify that the investor has not refunded
        _verifyHasNotRefunded(positionId);                    <<@
```

Proof of Concept (PoC):

1. `Investor1` and `Investor2` invest as per their SAFT.
2. `Investor2` decides to refund their position.
3. An OTC deal is struck between `Investor1` and `Investor2`.
4. The transfer leads to broken cached amounts for the `Investor2`'s position.

Add the following test case inside the `LegionCapitalRaiseTest.t.sol` file:

```
    function
test_transferInvestorPosition_successfullyTransfersInvestorPositionToExistingInvestor_breakingCachedValues() public {
        // Arrange
        prepareCreateLegionCapitalRaise();
        prepareMintAndApproveTokens();
        prepareInvestorSignatures();

        vm.warp(block.timestamp + 1);

        vm.prank(investor1);
        ILegionCapitalRaise(legionCapitalRaiseInstance).invest(
            10_000 * 1e6, 10_000 * 1e6, 5_000_000_000_000_000, signatureInv1
        );

        vm.prank(investor2);
        ILegionCapitalRaise(legionCapitalRaiseInstance).invest(
            10_000 * 1e6, 10_000 * 1e6, 5_000_000_000_000_000, signatureInv2
        );
```

```
        // Investor2 refunds their position
        vm.prank(investor2);
        ILegionCapitalRaise(legionCapitalRaiseInstance).refund();

        vm.prank(projectAdmin);
        ILegionCapitalRaise(legionCapitalRaiseInstance).end();

        vm.warp(block.timestamp + 2 weeks + 1);

        // Act
        vm.prank(legionBouncer);
        LegionCapitalRaise(legionCapitalRaiseInstance).transferInvestorPosition(investor1,
investor2, 1);

        // The total invested capital for investor2 is 10_000
        assertEq(

LegionCapitalRaise(payable(legionCapitalRaiseInstance)).investorPosition(investor2).investedCapital,
            10_000 * 1e6
        );


        // However, the cached token allocation rate considers the old `5_000_000_000_000_000`
cached allocation rate, even though the investor2 has refunded
        assertEq(
            LegionCapitalRaise(payable(legionCapitalRaiseInstance)).investorPosition(investor2)
                .cachedTokenAllocationRate,
            10_000_000_000_000_000      // Ideally, should be 5_000_000_000_000_000
        );

        // Similarly, the cached investment amount considers the burnt position's cached
investment amount
        assertEq(

LegionCapitalRaise(payable(legionCapitalRaiseInstance)).investorPosition(investor2).cachedInvestAmount,
            20_000 * 1e6                // Ideally, should be 10_000 * 1e6
        );
    }
```

Recommendations

It is recommended to clear and burn the position whenever a refund is processed.

[M-03] Transferred investment position is not claimable

Severity

Impact: High

Likelihood: Medium

Description

After the refund period ends and the sale results are published, Merkle proofs are set for each investor's accepted capital and allocated tokens. These proofs are tied to specific investment positions and investors.

If a position is transferred after this point, the transfer function does not update or change the Merkle proofs associated with the original position. Because the proofs remain linked to the investors, the receiver of the transferred position cannot claim the allocated tokens using the existing proofs.

This effectively blocks the new position owner from claiming tokens, causing loss of access to rightful token allocations after a transfer.

```
    function _verifyCanClaimTokenAllocation(
        address _investor,
        uint256 _amount,
        LegionVestingManager.LegionInvestorVestingConfig calldata _investorVestingConfig,
        bytes32[] calldata _proof
    )
        internal
        view
        virtual
    {
        // Get the investor position ID
        uint256 positionId = _getInvestorPositionId(_investor);

        // Verify that the position exists
        _verifyPositionExists(positionId);

        // Generate the merkle leaf
        bytes32 leaf =
@>          keccak256(bytes.concat(keccak256(abi.encode(_investor, _amount, positionId,
_investorVestingConfig))));

        // Load the investor position
        InvestorPosition memory position = s_investorPositions[positionId];

        // Verify the merkle proof
        if (!MerkleProofLib.verify(_proof, s_saleStatus.claimTokensMerkleRoot, leaf)) {
            revert Errors.LegionSale__NotInClaimWhitelist(_investor);
        }

        // Check if the investor has already settled their allocation
        if (position.hasSettled) revert Errors.LegionSale__AlreadySettled(_investor);
    }
```

Recommendations

Only allow transfer of investment positions before `refundEndTime` when the results are not calculated and published yet.

Low findings

[L-01] `LegionPositionManager` violates `ERC-721` by ignoring invalid tokens

The `LegionPositionManager` contract implements the ERC-721 standard but deviates from the specification in its `tokenURI()` function. According to the ERC-721 standard (EIP-721), the `tokenURI()` method must revert when a non-existent `tokenId` is passed. However, the current implementation does not include this validation, leading to a violation of the standard.

```
function tokenURI(uint256 tokenId) public view virtual override returns (string memory) {
    // Missing check for non-existent tokenId
    return string(abi.encodePacked(_baseURI(), _toString(tokenId)));
}
```

Add a check to ensure the token exists before returning the URI:

```
    if (!_exists(tokenId)) {
        revert("ERC721: URI query for nonexistent token");
    }
```

[L-02] `invest()` missing `_verifyCanClaimExcessCapital()` enables reinvestment

The `invest()` function in all sales contracts except `LegionPreLiquidApprovedSale` and `LegionCapitalRaise` has the `_verifyCanClaimExcessCapital()` check, which forbids investors from re-investing again after they have claimed excess funds.

It is recommended to add the `_verifyCanClaimExcessCapital()` for the `LegionPreLiquidApprovedSale` and `LegionCapitalRaise` contracts.

[L-03] `LegionBouncer` update relies on outdated bouncer post `ID` change

The admin is allowed to update the `LEGION_BOUNCER_ID` address via the `LegionAddressRegistry` contract using the `LegionAddressRegistry::setLegionAddress()` call. However, post update, syncing the addresses in `LegionCapitalRaise` along with all the sales contracts requires the old `legionBouncer` address.

```
    function syncLegionAddresses() external onlyLegion {                  <<@
        _syncLegionAddresses();
    }
```

The `onlyLegion` modifier still pertains to the last `legionBouncer` address:

```
    modifier onlyLegion() {
        if (msg.sender != s_saleConfig.legionBouncer) revert
Errors.LegionSale__NotCalledByLegion();
        _;
    }
```

Hence, it would be critical in case of `legionBouncer` being compromised in some way, or would increase the overhead of updating across all vesting contracts.

It is recommended to fetch the latest bouncer via the modifier call itself using the address registry to ensure no updates are required in case of an update.

[L-04] `emergencyTransferOwnership()` can be front-run to inaccessible wallets

The `emergencyTransferOwnership()` function is used in `LegionLinearVesting` and `LegionLinearEpochVesting` contracts to transfer the user's ownership in case they lose access to their wallets. However, a malicious actor can use this to grief the user by forcefully calling the `release()` function as it is public in nature.

```
    // File: LegionLinearEpochVesting.sol
    function release(address token) public override onlyCliffEnded {              <<@
```

Motivation behind the attacker here would be to lower the actual circulating supply, eventually helping the price of the token to stay afloat, usually around the cliff/epoch's end.

It is recommended to guard the `release()` function with the `onlyOwner` modifier.

[L-05] Changing `vestingController` not retroactive for contracts

The `_syncLegionAddresses` is used inside `LegionTokenDistributor.sol`, `LegionAbstractSale.sol`, and `LegionPreLiquidApprovedSale.sol` to ensure that the addresses set via `LegionAddressRegistry::setLegionAddress()` are updated.

However, this change does not apply to the current set of vesting contracts, which are already deployed. This would come into play whenever there's a need to transfer ownership in both `LegionLinearEpochVesting` and `LegionLinearVesting` contracts:

```
    function emergencyTransferOwnership(address newOwner) external onlyVestingController {
        if (newOwner == address(0)) {
            revert OwnableInvalidOwner(address(0));
        }
        _transferOwnership(newOwner);
    }
```

It is recommended to store the address registry inside the vesting contract and add a vesting controller sync function, which would allow for smoother sync.

[L-06] No expiration or invalidation controls for investment signatures

In the `LegionAbstractSale` contract, the `_verifyInvestSignature()` function is used in `invest` function to verify that an investor is authorized to invest. However, the current implementation is vulnerable to replay attacks because the signature verification does not include a nonce, deadline, or any way to invalidate the signature after it is issued.

```
    function _verifyInvestSignature(bytes calldata _signature) internal view virtual {
        bytes32 _data = keccak256(abi.encodePacked(msg.sender, address(this),
block.chainid)).toEthSignedMessageHash();

        if (_data.recover(_signature) != s_addressConfig.legionSigner) {
            revert Errors.LegionSale__InvalidSignature(_signature);
        }
    }
```

As a result, the signature remains valid forever and can be reused indefinitely. There is no way to cancel or revoke a signature once issued, increasing the risk in case of a leaked or compromised signature.

Recommendations

- Include a **nonce** in the signed message and track used nonces to prevent replay.
- Add a **deadline** field to limit how long a signature is valid.
- Store and track used **signature hashes** to block reuse. (if applicable).

[L-07] Paused `releaseVestedTokens()` can be bypassed

The releaseVestedTokens() allows users to collect their vested token as per the schedule and is used inside the `LegionTokenDistributor.sol`, `LegionAbstractSale.sol`, and `LegionPreLiquidApprovedSale.sol` contracts:

```
    function releaseVestedTokens() external whenNotPaused {
```

In case the contract is paused, the `whenNotPaused` is intended to disallow users from releasing their vested tokens. However, this can be directly bypassed as no such restrictions gets imposed on the `LegionLinearEpochVesting.sol` and `LegionLinearVesting.sol` contracts:

```
// File: LegionLinearVesting.sol

    function release(address token) public override onlyCliffEnded {
        super.release(token);
    }
```

```
// File: LegionLinearEpochVesting.sol

    function release(address token) public override onlyCliffEnded {
        super.release(token);

        // Update the last claimed epoch
        _updateLastClaimedEpoch();
    }
```

Proof of Concept (PoC):

1. Admin pauses the `LegionTokenDistributor`, `LegionPreLiquidApprovedSale` or any contract inheriting `LegionAbstractSale`.
2. Anyone can call the `LegionLinearVesting::release` or `LegionLinearEpochVesting::release` independently to bypass the pause.

Add this import on top of the `LegionTokenDistributor.t.sol` file:

```
import { ILegionVesting } from "../../src/interfaces/vesting/ILegionVesting.sol";
```

Finally, add the following test case:

```
    function test_bypass_releaseVestedTokens_whenPaused() public {
        // Arrange
        prepareCreateLegionTokenDistributor();
        prepareMintAndApproveTokens();
        prepareInvestorSignatures();

        vm.warp(block.timestamp + 1);

        vm.prank(projectAdmin);
        ILegionTokenDistributor(legionTokenDistributorInstance).supplyTokens(20_000 * 1e18, 500
* 1e18, 200 * 1e18);

        vm.prank(investor1);
        ILegionTokenDistributor(legionTokenDistributorInstance).claimTokenAllocation(
            uint256(5000 * 1e18), investorLinearVestingConfig, signatureInv1Claim,
vestingSignatureInv1
        );

        vm.warp(block.timestamp + 4 weeks + 1 hours + 3600);

        // Pause the contract
        vm.prank(legionBouncer);
        ILegionTokenDistributor(legionTokenDistributorInstance).pause();
        // Act
        vm.prank(investor1);
        // Should revert since the contract is paused
        vm.expectRevert();
        ILegionTokenDistributor(legionTokenDistributorInstance).releaseVestedTokens();

        // No change in balance
        assertEq(MockERC20(askToken).balanceOf(investor1), 500000000000000000000);

        address vestingAddress =
ILegionTokenDistributor(legionTokenDistributorInstance).investorPosition(investor1).vestingAddress;
```

```
        // Anyone can call release on the vesting contract even though the main contract is
paused
        vm.prank(investor1);
        ILegionVesting(vestingAddress).release(address(askToken));
        // Expect
        assertEq(MockERC20(askToken).balanceOf(investor1), 501_026_969_178_082_191_780);
    }
```

Recommendations

Consider adding a similar `Pausable` mechanism in the vesting contract, which only allows the vesting controller to pause the contract.