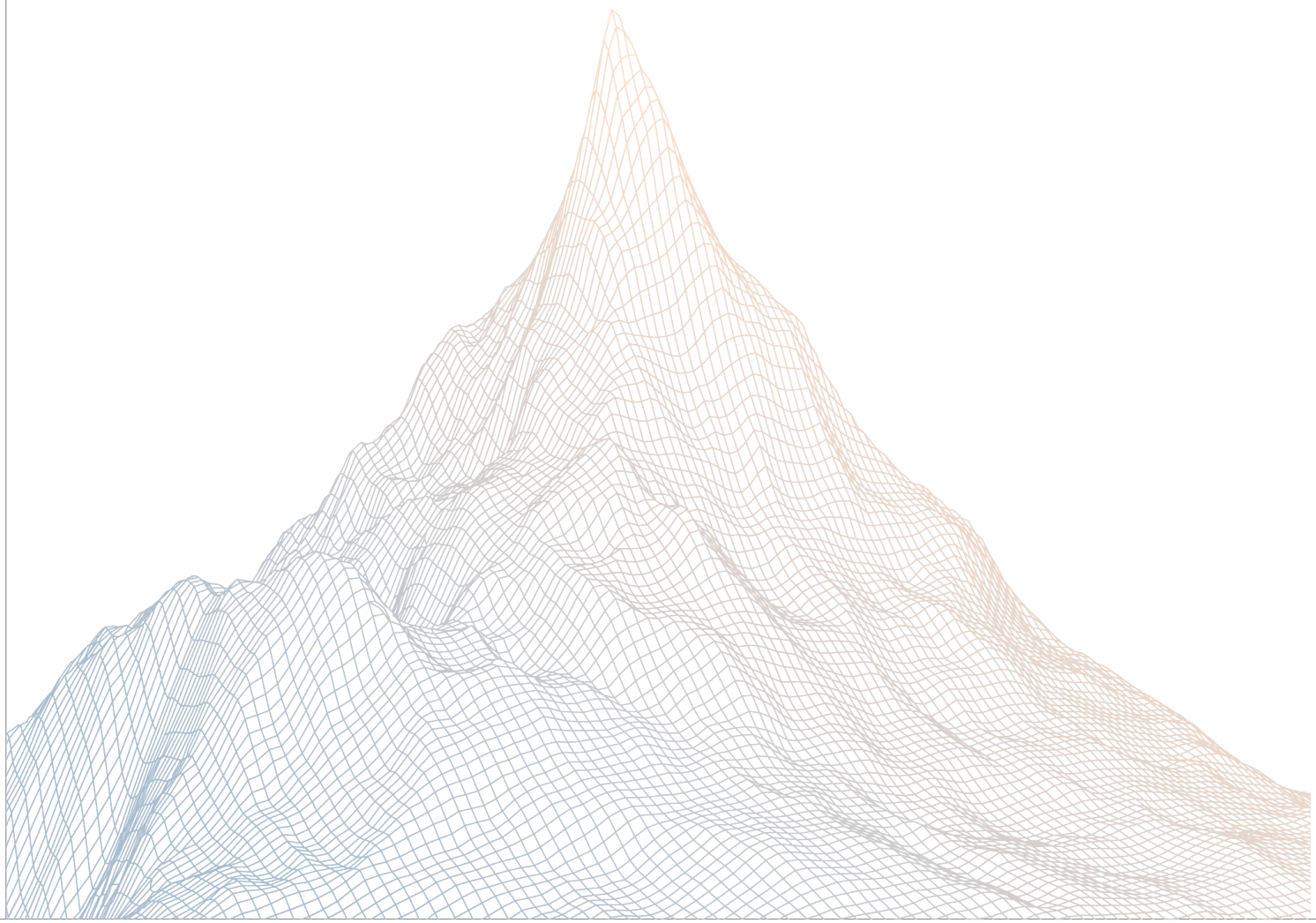
Zenith

Legion

Smart Contract Security Assessment

VERSION 1.1



AUDIT DATES: February 15nd to February 19th, 2025

AUDITED BY: matte
rscodes
spicymeatball

Contents

1 Introduction

1.1 About Zenith

Zenith is an offering by Code4rena that provides consultative audits from the very best security researchers in the space. We focus on crafting a tailored security team specifically for the needs of your codebase.

Learn more about us at https://code4rena.com/zenith.

1.2 Disclaimer

This report reflects an analysis conducted within a defined scope and time frame, based on provided materials and documentation. It does not encompass all possible vulnerabilities and should not be considered exhaustive.

The review and accompanying report are presented on an "as-is" and "as-available" basis, without any express or implied warranties.

Furthermore, this report neither endorses any specific project or team nor assures the complete security of the project.

1.3 Risk Classification

SEVERITY LEVEL	IMPACT: HIGH	IMPACT: MEDIUM	IMPACT: LOW
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

2

Executive Summary

2.1 About Legion

The goal of Legion is to create a network where anyone can freely chat and socialize without compromising their privacy, using the hashgraph consensus.

2.2 Scope

The engagement involved a review of the following targets:

Target	evm-contracts
Repository	https://github.com/Legion-Team/evm-contracts/
Commit Hash	`8c81309b71c64eb719f39fb25cf658015564223f`
Files	`LegionPreLiquidSaleV1.sol` `LegionPreLiquidSaleV2.sol`

2.3 Audit Timeline

DATE	EVENT
February 15, 2025	Audit start
February 19, 2025	Audit end
February 28, 2025	Report published

2.4 Issues Found

SEVERITY	COUNT
Critical Risk	0
High Risk	2
Medium Risk	6
Low Risk	3
Informational	3
Total Issues	**14**

3

Findings Summary

ID	DESCRIPTION	STATUS
H-1	Project admin can withdraw all invested funds, including excess capital	Resolved
H-2	Users can claim their ask tokens and steal back their bid tokens from the contract	Resolved
M-1	Refund and claiming stages can overlap	Resolved
M-2	Project owners are not required to return withdrawn capital upon sale cancellation	Resolved
M-3	‘publishTgeDetails‘ should set ‘refundEndTime‘ so that capital can’t be withdrawn	Resolved
M-4	Users unable to claim tokens if the sale was conducted on a different chain	Resolved
M-5	Project owners are not required to provide ask tokens before withdrawing capital	Acknowledged
M-6	An user who refunds and then buys again may not be able to refund	Resolved
L-1	Vesting parameters are not validated	Resolved
L-2	Impossible to withdraw excess tokens after a user has already withdrawn excess once	Resolved
L-3	Some functions should have the ‘whenNotPaused‘ modifier	Resolved
I-1	Users can provide stale allocation rate data when interacting with the sale contract	Acknowledged
I-2	Users can supply malformed signatures to bypass _verifySignatureNotUsed	Resolved
I-3	‘publishTgeDetails‘ should call ‘_verifyRefundPeriodIsOver‘	Resolved

4

Findings

4.1 High Risk

A total of 2 high risk findings were identified.

[H-1] Project admin can withdraw all invested funds, including excess capital

SEVERITY: High	IMPACT: High
STATUS: Resolved	LIKELIHOOD: Medium

Target

- LegionPreLiquidSaleV1.sol#L349

Description:

When the project admin calls the `withdrawRaisedCapital` function, they receive the entire amount invested by users, without distinguishing between actual capital and excess capital:

```
    function withdrawRaisedCapital() external onlyProject whenNotPaused {
        /// Verify that the sale is not canceled
        _verifySaleNotCanceled();

        /// Verify that the sale has ended
        _verifySaleHasEnded();

        // Verify that the refund period is over
        _verifyRefundPeriodIsOver();

        /// Verify that the project can withdraw capital
        _verifyCanWithdrawCapital();

        /// Account for the capital withdrawn
>>      saleStatus.totalCapitalWithdrawn = saleStatus.totalCapitalInvested;
```

As a result, users who invested more than the required amount would be unable to withdraw their excess capital, since no bid tokens would remain in the contract.

Recommendations:

Consider transferring only the actual capital to the project admin, similar to how it is handled in the V2 sale:

```
function publishCapitalRaised(uint256 capitalRaised,
    bytes32 acceptedMerkleRoot) external onlyLegion {
    --- SNIP ---

    // Set the total capital raised to be withdrawn by the project
    saleStatus.totalCapitalRaised = capitalRaised;

function withdrawRaisedCapital() external override(ILegionSale, LegionSale)
    onlyProject whenNotPaused {
    --- SNIP ---

    // Cache value in memory
    uint256 _totalCapitalRaised = saleStatus.totalCapitalRaised;
```

Legion: Fixed in @182e9e02c8....

Zenith: Verified. The project admin can withdraw only the actual raised capital published by the Legion protocol.

[H-2] Users can claim their ask tokens and steal back their bid tokens from the contract

SEVERITY: High	IMPACT: High
STATUS: Resolved	LIKELIHOOD: Medium

Target

- LegionPreLiquidSaleV1.sol

Description:

The pre-liquid sale ends when `saleStatus.hasEnded` is set to `true`.

And there are 2 functions that can set `saleStatus.hasEnded` to true and end the sale.

- Either call `endSale` or call `publishTgeDetails`.

Both are **coded to end the sale by setting the value of** *`saleStatus.hasEnded`*. However, taking a closer look at `endSale`, we can see that `endSale` calls `_verifySaleHasNotEnded()`

- Which means that once `publishTgeDetails` is called, then `endSale` **cannot** be called. Since `hasEnded` will be true and `endSale` will **revert**.

However, we can see that `endSale` is the **only** function that can set `saleStatus.refundEndTime`. Which means that if `endSale` cannot be called then `refundEndTime` will **forever** be 0.

Now, let's look at what happens if `refundEndTime` is 0. If it is zero, it means that both `_verifyRefundPeriodIsOver` and `_verifyRefundPeriodIsNotOver` will pass.

So after `publishTgeDetails` and `supplyAskTokens`, what the attacker can now do is:

- Call `claimAskTokenAllocation` and get their ask tokens. (However, `claimAskTokenAllocation` **does not** reset `positions.investedCapital` to 0.
- Call `refund` which will not revert as `_verifyRefundPeriodIsNotOver()` will pass due to what was described above. And attacker **steals back their bid token** becoz positions.investedCapital is still the original value.

So now the attacker managed to get both the bid and ask tokens.

Recommendations:

Set `positions.investedCapital = 0` in the function `claimAskTokenAllocation`. (And also allow `endSale` to be called even if `hasEnded` is true, so that `refundEndTime` can be rightfully set)

Legion: Fixed with @819765e408.... We decided to take the approach that publishTgeDetails is only callable after the sale has been ended by Legion or the Project. So, the only way to end the sale will be through the endSale method.

Zenith: Verified

4.2 Medium Risk

A total of 6 medium risk findings were identified.

[M-1] Refund and claiming stages can overlap

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Low

Target

- LegionSale.sol#L183

Description:

In the V1 sale, the following conditions must be met for users to claim their tokens:

- The sale is not canceled.
- TGE data has been published.
- Ask tokens have been supplied by the project.

However, nothing prevents the protocol from satisfying all these conditions while the refund period is still active. This means users could claim their ask tokens and then refund their initial investment, potentially exploiting the system.

Recommendations:

Ensure that the refund period has ended before allowing TGE data to be published.

Legion: Fixed in @67ae4c3966...

Zenith: Verified. The protocol will ensure the refund period is over before publishing the TGE.

[M-2] Project owners are not required to return withdrawn capital upon sale cancellation

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Low

Target

- LegionPreLiquidSaleV2.sol#L187

- LegionSale.sol#L357

Description:

If a project owner withdraws capital from the sale and later decides to cancel it, they are not obligated to return the withdrawn tokens:

```
function cancelSale() public virtual onlyProject whenNotPaused {
    // Allow the Project to cancel the sale at any time until results are
    published
    // Results are published after the refund period is over
    _verifySaleResultsNotPublished();

    // Verify sale has not already been canceled
    _verifySaleNotCanceled();

    // Mark sale as canceled
    saleStatus.isCanceled = true;

    // Emit successfully SaleCanceled
    emit SaleCanceled();
}
```

The only restriction is that the sale results must not have been published. However, since project owners can withdraw bid tokens once Legion has published the raised capital value, nothing prevents them from taking the funds and then canceling the sale.

Recommendations:

Consider implementing a mechanism to retrieve withdrawn tokens from the project admin when `cancelSale` is called to ensure fairness and protect investors.

Legion: Fixed in @15b46079c2....

Zenith: Verified - Canceling the sale requires returning withdrawn raised capital if it was taken.

[M-3] `publishTgeDetails` should set `refundEndTime` so that capital can't be withdrawn

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Low

Target

- LegionPreLiquidSaleV1.sol

Description:

According to the docs, users are entitled to a refund period.

During an on-going refund period, `project` is strictly not allowed to call `withdrawRaisedCapital` as the investors are entitled by the rules to refund the current bid tokens sitting in the contract.

- This can be seen from `withdrawRaisedCapital` having a `_verifyRefundPeriodIsOver` check.

This makes sense as only the admin `legion` is fully trusted and the external `project` will have to **abide by the TGE rules** set by the admin `legion` which is also explained in my previous submission: here on why only legion is fully trusted and not project.

Right now, there are 2 ways to end the sale, (end the sale means `saleStatus.hasEnded = true`). The 2 ways are either `publishTgeDetails` or `endSale`. And it is worth noting once `publishTgeDetails` is called, `endSale` cannot be called anymore as `endSale` requires `saleStatus.hasEnded` to be false. But on the surface, there is also no reason for `endSale` to be called anymore once `publishTgeDetails` is called as `publishTgeDetails` will end the sale by setting `saleStatus.hasEnded = true`

Looking at `publishTgeDetails`:

```
function publishTgeDetails(
    address _askToken,
    uint256 _askTokenTotalSupply,
    uint256 _vestingStartTime,
    uint256 _totalTokensAllocated
)
    external
    onlyLegion
{
    ....
```

```
    /// Set `hasEnded` status to true
--> if (!saleStatus.hasEnded) saleStatus.hasEnded = true;

    ....
}
```

We can see that `publishTgeDetails` ends the sale, but **does not** set the refund time. This means that if the sale is ended through `publishTgeDetails` instead of `endSale` then **refundEndTime will just remain as zero forever.** (note that once the sale is ended through `publishTgeDetails` then `endSale` **can't be called anymore**)

refundEndTime being zero is perfectly normal for `refund` as users can still have their entitled refund period because `_verifyRefundPeriodIsNotOver` will pass. **However, the edge case of refundEndTime being zero is that** ***`_verifyRefundPeriodIsOver`*** **will also pass**. Meaning `withdrawRaisedCapital` can be called, and some users may not be able to refund.

Recommendations: Set `saleStatus.refundEndTime = block.timestamp + saleConfig.refundPeriodSeconds;` in `publishTgeDetails` if `!saleStatus.hasEnded` as that means endSale hasn't been called.

Legion: Resolved with @819765e408...

Zenith: Verified

[M-4] Users unable to claim tokens if the sale was conducted on a different chain

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Medium

Target

- LegionSale.sol#L603

Description:

Sales can be conducted across different chains, such as raising capital on chain A and distributing tokens on chain B. In this scenario, users would be unable to claim their ask tokens due to the following check:

```
    function _verifyCanClaimTokenAllocation(
        address _investor,
        uint256 _amount,
        bytes32[] calldata _proof
    )
        internal
        view
        virtual
    {
        --- SNIP ---

        // Safeguard to check if the investor has pledged capital
>>      if (position.investedCapital == 0)
    revert Errors.NoCapitalInvested(_investor);
    }
```

Since users may have invested on a different chain, their `investedCapital` value in this contract would be 0, preventing them from claiming their tokens.

Recommendations:

Consider relaxing the eligibility criteria to allow users to claim tokens even if they do not have capital recorded in the current sale contract.

Legion: Fixed in 1c46cc8293e

Zenith: Verified. Users can claim tokens without having capital invested in the current sale.

[M-5] Project owners are not required to provide ask tokens before withdrawing capital

SEVERITY: Medium	IMPACT: Medium
STATUS: Acknowledged	LIKELIHOOD: Low

Target

- LegionPreLiquidSaleV1.sol#L335
- LegionPreLiquidSaleV2.sol#L187

Description:

Unlike the base sale contract, which requires project owners to supply ask tokens before withdrawing capital:

```
    function withdrawRaisedCapital()
    external virtual onlyProject whenNotPaused {
        --- SNIP ---

        // Check if projects are withdrawing capital on the sale source chain
        if (addressConfig.askToken ≠ address(0)) {
            // Allow projects to withdraw capital only in case they've
    supplied tokens
>>          _verifyTokensSupplied();
        }
```

V1 and V2 sales do not enforce this requirement. As a result, project owners can withdraw bid tokens from the contract without ever supplying ask tokens. This creates a potential risk where project owners could take users' funds without providing anything in return, ultimately undermining trust in the protocol.

Recommendations:

Implement a check to ensure that ask tokens are supplied before allowing project owners to withdraw capital.

Legion: Acknowledged. This is intentional by design for both versions of the pre-liquid sales. As there could be a significant time gap between raising capital and the Token Generation Event (TGE), projects are allowed to withdraw the raised funds before supplying the tokens sold to the investors.

[M-6] An user who refunds and then buys again may not be able to refund

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Medium

Target

- LegionPreLiquidSaleV1.sol

Description:

According to the code, an user can refund when the sale hasn't ended. This is because, if the sale hasn't ended, it means that `endSale` **has not been called**. If it hasn't been called, **it means that** *`refundEndTime`* **is still 0**.

Since `refundEndTime` is still 0, `_verifyRefundPeriodIsNotOver` will not revert, hence `refund` **can be called** during the sale.

So consider this senario of `Alice` who has bought tokens from the sale.

1. Alice decides to refund previously bought tokens
2. Before the sale ends later, Alice decides to buy tokens again (possibly a different amount this time which would explain the change of decision to buy tokens again)
3. `endSale` is called, and `refundEndTime` is set, officially beginning the refund period.
4. However, now Alice cannot refund during the refund period as `refund` calls `_verifyHasNotRefunded()`

Recommendations:

`invest` should just set `investorPositions[msg.sender].hasRefunded` to `false`. That way edge cases like this can be prevented.

Legion: Resolved with @abaabfa82a...

Zenith: Verified.

4.3 Low Risk

A total of 3 low risk findings were identified.

[L-1] Vesting parameters are not validated

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- LegionPreLiquidSaleV1.sol#L290

Description:

The project admin can override vesting parameters that were initially set during the sale initialization:

```
function updateVestingTerms(
    uint256 _vestingDurationSeconds,
    uint256 _vestingCliffDurationSeconds,
    uint256 _tokenAllocationOnTGERate
)
    external
    onlyProject
    whenNotPaused
{
```

This creates a potential risk where a malicious project owner could set an extremely long vesting duration, effectively preventing users from claiming their tokens.

Recommendations:

Implement validation checks in `updateVestingTerms` to ensure that vesting parameters remain reasonable.

Legion: Fixed in @7248e45d260....

Zenith: Verified. Vesting parameters are now checked during updates and initialization.

[L-2] Impossible to withdraw excess tokens after a user has already withdrawn excess once

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- LegionSale.sol#L626

Description:

The Legion protocol allows setting the accepted capital Merkle root at any time during the sale:

```
function setAcceptedCapital(bytes32 merkleRoot)
    external virtual onlyLegion {
    // Verify that the sale is not canceled
    _verifySaleNotCanceled();

    // Verify that the sale has not ended
    _verifySaleHasNotEnded();

    // Set the merkle root for accepted capital
    saleStatus.acceptedCapitalMerkleRoot = merkleRoot;

    // Emit successfully AcceptedCapitalSet
    emit AcceptedCapitalSet(merkleRoot);
}
```

This creates a rare scenario where a user withdraws excess capital and then reinvests. If they generate excess capital again, they will be unable to withdraw it because the contract only allows excess capital withdrawals once:

```
    function _verifyCanClaimExcessCapital(
        address _investor,
        uint256 _amount,
        bytes32[] calldata _proof
    )
        internal
        view
        virtual
    {
        // Load the investor position
        InvestorPosition memory position = investorPositions[_investor];

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

Recommendations:

Restrict users from reinvesting after they have withdrawn excess capital to prevent this issue.

Legion: Fixed in @bea8583f3c....

Zenith: Verified. Users cannot reinvest if they already hold excess tokens.

[L-3] Some functions should have the `whenNotPaused` modifier

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- LegionPreLiquidSaleV1.sol
- LegionPreLiquidSaleV2.sol

Description:

- `publishTgeDetails` from `LegionPreLiquidSaleV1.sol`
- `publishCapitalRaised` from `LegionPreLiquidSaleV2.sol`
- `publishSaleResults` from `LegionPreLiquidSaleV2.sol`

The 3 functions above does not have the `whenNotPaused` modifier. Since these 3 functions are not related at all to the emergency rescue function (which yep should not have the modifier), they should include the `whenNotPaused` modifier just like the rest of the non-emergency functions.

Recommendations:

Add `whenNotPaused` to those 3 non emergency functions which shouldn't be called during an emergency paused market.

Legion: Resolved with @9b2247e0d3e...

Zenith: Verified.

4.4 Informational

A total of 3 informational findings were identified.

[I-1] Users can provide stale allocation rate data when interacting with the sale contract

SEVERITY: Informational	IMPACT: Informational
STATUS: Acknowledged	LIKELIHOOD: Low

Target

- LegionPreLiquidSaleV1.sol#L148
- LegionPreLiquidSaleV1.sol#L549
- LegionPreLiquidSaleV1.sol#L399

Description:

When a user performs any of the following actions:

- Investing bid tokens
- Withdrawing excess bid tokens
- Claiming ask tokens

they provide data that is calculated off-chain. One of these parameters is `tokenAllocationRate`:

```
    function invest(
        uint256 amount,
        uint256 investAmount,
        uint256 tokenAllocationRate,
        bytes32 saftHash,
        bytes memory signature
    )
        external
        whenNotPaused
    {
        --- SNIP ---
        /// Cache the token allocation rate in 18 decimals precision
>>      if (position.cachedTokenAllocationRate != tokenAllocationRate) {
            position.cachedTokenAllocationRate = tokenAllocationRate;
        }
```

Currently, the sale contract does not include any checks to prevent users from specifying stale rates. For example, if a user receives a signature and allocation rate from the UI but waits a week before calling the `invest` function, the rates will likely have changed. However, the outdated rate would still be stored in the contract upon execution.

Recommendations:

If the cached position values are used outside of the sale contract, it is recommended to add an expiry parameter in the signature and user-provided data. This should be validated alongside other parameters in `_verifyValidPosition` to ensure the provided allocation rate is up to date.

Legion: Acknowledged. When a user receives a signature generated by Legion, the `tokenAllocationRate` is fixed and calculated off-chain, relative to the `investAmount` and FDV, meaning it doesn't change during the sale.

[I-2] Users can supply malformed signatures to bypass _verifySignatureNotUsed

`SEVERITY:` Informational	`IMPACT:` Informational
`STATUS:` Resolved	`LIKELIHOOD:` Low

Target

- LegionPreLiquidSaleV1.sol#L840
- LegionPreLiquidSaleV1.sol#L906

Description:

The `LegionPreLiquidSaleV1` contract uses Solady's ECDSA library to recover and validate the signer:

```
    function _verifyValidPosition(bytes memory signature,
    SaleAction actionType) internal view {
         --- SNIP ---
        /// Construct the signed data
        bytes32 _data = keccak256(
            abi.encodePacked(
                msg.sender,
                address(this),
                block.chainid,
                uint256(position.cachedInvestAmount),
                uint256(position.cachedTokenAllocationRate),
                bytes32(uint256(position.cachedSAFTHash)),
                actionType
            )
```

```
        ).toEthSignedMessageHash();

        /// Verify the signature
>>      if (_data.recover(signature) != saleConfig.legionSigner)
    revert Errors.InvalidSignature();
    }
```

However, as noted in the comments, Solady's ECDSA library does not check whether the signature is malformed.This allows an attacker to bypass uniqueness checks, such as the one performed in `_verifySignatureNotUsed`, by submitting a malformed signature and executing the same action again:

```
function _verifySignatureNotUsed(bytes memory signature) private view {
    /// Check if the signature is used
    if (usedSignatures[msg.sender][signature])
    revert Errors.SignatureAlreadyUsed(signature);
}
```

Recommendations:

Although this issue does not pose an immediate threat due to other guard checks preventing exploitation, it is recommended to use OpenZeppelin's ECDSA library. This library includes built-in checks to prevent the use of malformed signatures, improving security and reliability

Legion: Fixed in @73352065f3....

Zenith: Verified.

[I-3] `publishTgeDetails` should call `_verifyRefundPeriodIsOver`

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- LegionPreLiquidSaleV1.sol

Description:

In `LegionPreLiquidSaleV1.sol`, the only check that `publishTgeDetails` calls is `_verifySaleNotCanceled();`

However, if users are still refunding and some of the bid tokens are leaving the contract, then `publishTgeDetails` shouldn't be called yet since `saleStatus.askTokenTotalSupply` should be adjusted according to the bidders and total bid token amounts from the users.

Recommendations:

```
function publishTgeDetails(
    address _askToken,
    uint256 _askTokenTotalSupply,
    uint256 _vestingStartTime,
    uint256 _totalTokensAllocated
)
    external
    onlyLegion
{
    /// Verify that the sale has not been canceled
    _verifySaleNotCanceled();
   _verifyRefundPeriodIsOver();
    ....
}
```

Legion: Fixed with @67ae4c3966...

Zenith: Verified