

Legion Findings & Analysis Report

Published:Oct 05, 2024Prepared for:LegionPrepared by:Code4renaAudit dates:Sep 10—Sep 17, 2024



Contents

1. Overview

- 1.1 About C4
- 1.2 About Legion

2. Summary

3. Scope

4. Severity Criteria

5. Audit Timeline

6. High Risk Findings

6.1. Incorrect formula used for totalCapitalRaised calculation in publishSaleResults function

6.2. withdrawCapital() can be called multiple times by the project to withdraw ExcessCapital

6.3. when cachedTokenAllocationBps decreases , investors can maliciously use the old cachedTokenAllocationBps before refund

6.4. withdrawRaisedCapital() the project can malicious withdrawals

7. Medium Risk Findings

7.1. Legion signature can be reused

7.2. withdrawCapitallfSaleIsCanceled() Investors who have already settled can still refund

8. Low Risk Findings

8.1. Impossible to update Legion addresses when the sale is initialized

9. Informational Findings

- 9.1. Misplaced values in saleConfiguration() function
- 9.2. ECIES#isValid() Validity check is not complete



1. Overview

1.1 About C4

Code4rena (C4) is an open organization consisting of security researchers, auditors, developers and individuals with domain expertise in smart contracts.

A C4 audit is an event in which community participants, referred to as Wardens, review, audit or analyze smart contract logic in exchange for a bounty provided by sponsoring projects

During the audit outlined in this document, C4 conducted an analysis of the Legion smart contract system written in Solidity. The audit took place from Sep 10 to Sep 17, 2024.

1.2 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. Summary

SEVERITY	COUNT
Critical	0
High	4
Medium	2
Low	1
Informational	2

3. Scope

The source code was delivered to Code4rena in a private Git repository.



4. Severity Criteria

C4 assesses the severity of disclosed vulnerabilities based on the primary risk categories: high, medium, low and informational.

High-level considerations for vulnerabilities span the following key areas when conducting assessments:

- Malicious Input Handling
- Escalation of privileges
- Arithmetic
- Gas use

For more information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on <u>the C4</u> <u>website</u>, specifically our section on <u>Severity Categorization</u>.

5. Audit Timeline

DATE	EVENT
Sep 10, 2024	Kick-off call
Sep 10, 2024	Audit start
Sep 17, 2024	Audit end



6. High Risk Findings

A total of 4 high risk findings were identified.

6.1. Incorrect formula used for totalCapitalRaised calculation in publishSaleResults function

Severity: High	Status: Resolved	

Context:

LegionFixedPriceSale.sol#L157

Description:

After the sale is completed, the publishSaleResults function is called to configure the following parameters:

- totalTokensAllocated the total amount of ask tokens available for users
- totalCapitalRaised the amount of bid tokens that can be withdrawn by the project owners

```
function publishSaleResults(bytes32 merkleRoot, uint256
tokensAllocated) external onlyLegion {
    /// Verify that the sale is not canceled
    _verifySaleNotCanceled();
    /// Verify that the refund period is over
    _verifyRefundPeriodIsOver();
    /// Verify that sale results are not already published
    _verifyCanPublishSaleResults();
    /// Set the merkle root for claiming tokens
    claimTokensMerkleRoot = merkleRoot;
    /// Set the total tokens to be allocated by the Project
team
    totalTokensAllocated = tokensAllocated;
```



```
/// Set the total capital raised to be withdrawn by the
project
>> totalCapitalRaised = (tokensAllocated * tokenPrice) / (10
** (ERC20(bidToken).decimals()));
    /// Emit successfully SaleResultsPublished
    emit SaleResultsPublished(merkleRoot, tokensAllocated);
}
```

However, the formula for totalCapitalRaised can produce incorrect results when the ask and bid tokens have different decimal places. Consider the following scenario:

- There is a fixed price sale of VIP tokens (18 decimals) for USDC (6 decimals) at a price of \$3000 per VIP token.
- If 10 VIP tokens (10e18) are allocated, the total capital should be \$30,000 (30,000e6).
- However, using the current formula in the contract, we get 10 * 1e18 * 3000 * 1e6 / 1e6 = 30,000e18 USDC, which is incorrect.

Due to the inflated totalCapitalRaised, the project owner would be unable to withdraw the correct amount of bid tokens collected by the sale contract.

Recommendation:

The divisor should be 10 ****** (ERC20(askToken).decimals()) for correct calculation.

Legion:

The issue has been fixed with commit

C4 Zenith:

Divisor has been changed to 10 ****** (askTokenDecimals), where askTokenDecimals passed as an argument by the Legion caller.



6.2. withdrawCapital() can be called multiple times by the project to withdraw ExcessCapital

Severity: High

Status: Resolved

Context:

• LegionBaseSale.sol#L210

Description:

The project can take the raised capital totalCapitalRaised by withdrawCapital()

```
function withdrawCapital() external virtual onlyProject {
        /// Verify that the refund period is over
        _verifyRefundPeriodIsOver();
        /// Verify that the sale is not canceled
        _verifySaleNotCanceled();
        /// Verify that sale results have been published
        _verifySaleResultsArePublished();
       /// Check if projects are withdrawing capital on the sale
source chain
       if (askToken != address(0)) {
            /// Allow projects to withdraw capital only in case
they've supplied tokens
            _verifyTokensSupplied();
        }
        /// Cache value in memory
        uint256 _totalCapitalRaised = totalCapitalRaised;
        /// Calculate Legion Fee
        uint256 _legionFee = (legionFeeOnCapitalRaisedBps *
_totalCapitalRaised) / 10000;
        /// Emit successfully CapitalWithdrawn
        emit CapitalWithdrawn(_totalCapitalRaised, msg.sender);
```



The problem with this method is that it does not change the flag that prevents the project from executing multiple times.

The capital deposited in the contract is usually larger than totalCapitalRaised, and contains a portion of ExcessCapital that has not yet been withdrawn by the investor.

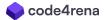
This part of the capital can be maliciously withdrawn by the project by executing withdrawCapital() multiple times.

Recommendation:

Add flag tokensWithdrawn.

```
abstract contract LegionBaseSale is ILegionBaseSale, Initializable {
    ...
    /// @dev Whether tokens have been supplied by the project or
not.
    bool internal tokensSupplied;
+ bool internal tokensWithdrawn;

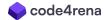
function withdrawCapital() external virtual onlyProject {
    /// Verify that the refund period is over
    _verifyRefundPeriodIsOver();
    /// Verify that the sale is not canceled
    _verifySaleNotCanceled();
    /// Verify that sale results have been published
_verifySaleResultsArePublished();
```



+	if (tokensWithdrawn) revert TokensAlreadyWithdrawn;
+	tokensWithdrawn = true;

Legion: The issue has been fixed with the following commit

C4 Zenith: The issue has been resolved by add capitalWithdrawn flag



6.3. when cachedTokenAllocationBps decreases , investors can maliciously use the old cachedTokenAllocationBps before refund

Severity: High

Status: Resolved

Context:

- LegionPreLiquidSale.sol#L416
- LegionPreLiquidSale.sol#L512

Description:

the project can modify the terms by updatingVestingTerms() and updateSAFTMerkleRoot().

For example, it is possible to reduce an investor's investment percentage, and after updating the terms, that investor can get back the ExcessCapital by withdrawExcessCapital().

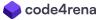
But currently there is no restriction on the order of claimAskTokenAllocation() and withdrawExcessCapital().

This way, a malicious investor can execute claimAskTokenAllocation() first and use the larger cachedTokenAllocationBps to maliciously obtain more askTokens.

After that, execute withdrawExcessCapital() to retrieve the ExcessCapital.

Example:

- 1. project call updateSAFTMerkleRoot()
 - cachedTokenAllocationBps[alice]= 5%
 - cachedSAFTInvestAmount = 500e6
- 2. alice call invest(500e6)
- 3. project call updateSAFTMerkleRoot()
 - cachedTokenAllocationBps[alice] = 1% ----> cachedTokenAllocationBps decreases
 - cachedSAFTInvestAmount = 100e6
- 4. alice call claimAskTokenAllocation()
 - get cachedTokenAllocationBps[alice]= 5% ask token



- 5. alice call withdrawExcessCapital()
 - get back 400e6 bidToken

Recommendation:

claimAskTokenAllocation() add in proof[] and validate saftMerkleRoot again.

Legion:

The issue has been fixed with the following commit

C4 Zenith:

The issue has been resolved by add proof param



6.4. withdrawRaisedCapital() the project can malicious withdrawals

Severity: High

Status: Resolved

Context:

• LegionPreLiquidSale.sol#L391

Description:

The project can uses withdrawRaisedCapital() to take away RaisedCapital.

```
function withdrawRaisedCapital(address[] calldata investors)
external onlyProject returns (uint256 amount) {
        /// Verify that the sale is not canceled
        _verifySaleNotCanceled();
        /// Loop through the investors positions
        for (uint256 i = 0; i < investors.length; ++i) {</pre>
            /// Verify that the refund period is over for the
specified position
            _verifyRefundPeriodIsOver(investors[i]);
            /// Verify that the investor has actually invested
capital
            _verifyCanWithdrawInvestorPosition(investors[i]);
            /// Load the investor position
            InvestorPosition storage position =
investorPositions[investors[i]];
            /// Mark the amount of capital withdrawn
(d)
            position.withdrawnCapital += position.investedCapital;
            /// Increment the total amount to be withdrawn
            amount += position.investedCapital;
        }
        /// Account for the capital withdrawn
        totalCapitalWithdrawn += amount;
```



```
/// Calculate Legion Fee
uint256 legionFee = (legionFeeOnCapitalRaisedBps * amount)
/ 10000;
/// Emit successfully CapitalWithdrawn
emit CapitalWithdrawn(amount);
/// Transfer the amount to the Project's address
IERC20(bidToken).safeTransfer(msg.sender, (amount -
legionFee));
/// Transfer the Legion fee to the Legion fee receiver
address
if (legionFee != 0)
IERC20(bidToken).safeTransfer(legionFeeReceiver, legionFee);
}
```

The problem is that the above method uses position.withdrawnCapital += position.investedCapital, instead of adding the difference between the two variables.

And since the terms are modifiable, i.e. cachedSAFTInvestAmount can be changed, this gives the project the opportunity to withdraw in advance and malicious transfer ExcessCapital

Example:

- 1. alice's cachedSAFTInvestAmount = 10
- 2. alice call invest(10)
- 3. the project change alice's cachedSAFTInvestAmount = 20
- 4. alice call withdrawRaisedCapital() ***front-run publishTgeDetails()
 - withdrawnCapital = 10
- 5. alice invest(10)->investedCapital +=10 ***front-run publishTgeDetails()
 - investedCapital = 20
- 6. leginon execute publishTgeDetails()
- 7. After that, the project executes withdrawRaisedCapital(alice) any times.
 - withdrawnCapital +=investedCapital = 10 + 20 = 30

Recommendation:

use the difference between the two variables

```
function withdrawRaisedCapital(address[] calldata investors)
external onlyProject returns (uint256 amount) {
        /// Verify that the sale is not canceled
        _verifySaleNotCanceled();
        /// Loop through the investors positions
        for (uint256 i = 0; i < investors.length; ++i) {</pre>
            /// Verify that the refund period is over for the
specified position
            _verifyRefundPeriodIsOver(investors[i]);
            /// Verify that the investor has actually invested
capital
            _verifyCanWithdrawInvestorPosition(investors[i]);
            /// Load the investor position
            InvestorPosition storage position =
investorPositions[investors[i]];
            /// Mark the amount of capital withdrawn
            position.withdrawnCapital += position.investedCapital;
            uint256 currentAmount = position.investedCapital -
+
position.withdrawnCapital;
            position.withdrawnCapital += currentAmount;
+
            /// Increment the total amount to be withdrawn
            amount += position.investedCapital;
+
            amount += currentAmount;
        }
```

Legion:

The issue has been fixed with the following commit

C4 Zenith:

The issue has been resolved as per recommendation



7. Medium Risk Findings

A total of 2 medium risk findings were identified.

7.1. Legion signature can be reused

```
Severity: Medium
```

Status: Resolved

Context:

- LegionBaseSale.sol#L652-L655
- LegionFixedPriceSale.sol#L106-L107
- LegionSealedBidAuction.sol#L98-L99

Description:

The sale contract verifies if a user is eligible to pledge capital by checking if the signature is signed by the trusted Legion signer:

In the _verifyLegionSignature function, only the caller's address is used in the hash:

This makes it possible to reuse the signature across all ongoing and future Legion sales. Moreover, the same signature can be used to gain investment access in



different chains where Legion is deployed.

Recommendation:

Consider including additional parameters in the hash so the signature is valid only for the specific sale:

Legion:

The issue has been fixed with commit

C4 Zenith:

The issue has been resolved as per recommendation.



7.2. withdrawCapitallfSaleIsCanceled() Investors who have already settled can still refund

Severity: Medium

Status: Resolved

Context:

• LegionPreLiquidSale.sol#L486

Description:

If the project owner decides to cancel PreLiquid, it can be done through the method cancelSale(). When canceling, need to return the withdrawn capital: totalCapitalWithdrawn. The investor can then retrieve the investment via withdrawCapitalIfSaleIsCanceled().

The problem is that currently withdrawCapitalIfSaleIsCanceled() doesn't restrict the refund of already settled investments. This can result in the investor receiving both askToken + refund bidToken.

Example:

- 1. alice invest 100 bidToken
- 2. publishTgeDetails() && supplyAskTokens()
- 3. the project call withdrawRaisedCapital(), totalCapitalWithdrawn = 1000
- 4. alice claimAskTokenAllocation(), get 100 askToken
- 5. the project Decide to cancel , call cancelSale() , return totalCapitalWithdrawn = 1000
- 6. alice call withdrawCapitalIfSaleIsCanceled() get 100 bidToken

so alice get 100 refund bidToken + 100 askToken

Recommendation:

It is recommended that what has been settled cannot be refunded, and the excess bidToken is returned to the project offline via emergencyWithdraw().

function withdrawCapitalIfSaleIsCanceled() external {



```
/// Verify that the sale has been actually canceled
_verifySaleIsCanceled();
/// Cache the amount to refund in memory
uint256 amountToClaim =
investorPositions[msg.sender].investedCapital;
+ if (investorPositions[msg.sender].hasSettled) revert
AlreadySettled(msg.sender);
```

Legion:

The issue has been fixed with the following commit

C4 Zenith:

The issue has been resolved It has been supplied and cannot be cancelled



8. Low Risk Findings

A total of 1 low risk finding was identified.

8.1. Impossible to update Legion addresses when the sale is initialized

Severity: Low	Status: Resolved	
ocverny. Low	oldida. Resolved	

Context:

LegionFixedPriceSale.sol#L95-L99 LegionSealedBidAuction.sol#L87-L91 LegionPreLiquidSale.sol#L161-L164

Description:

It's impossible to change addresses from the addressRegistry that were set during the sale initialization:

```
function initialize(FixedPriceSaleConfig calldata
fixedPriceSaleConfig) external initializer {
        ---SNIP---
        /// Calculate and set prefundStartTime, prefundEndTime,
startTime, endTime and refundEndTime
        prefundStartTime = block.timestamp;
        prefundEndTime = prefundStartTime +
fixedPriceSaleConfig.prefundPeriodSeconds;
        startTime = prefundEndTime +
fixedPriceSaleConfig.prefundAllocationPeriodSeconds;
        endTime = startTime +
fixedPriceSaleConfig.salePeriodSeconds;
        refundEndTime = endTime +
fixedPriceSaleConfig.refundPeriodSeconds;
        /// Check if lockupPeriodSeconds is less than
refundPeriodSeconds
        /// lockupEndTime should be at least refundEndTime
        if (fixedPriceSaleConfig.lockupPeriodSeconds <=</pre>
```

```
fixedPriceSaleConfig.refundPeriodSeconds) {
```

```
/// If yes, set lockupEndTime to be refundEndTime
            lockupEndTime = refundEndTime;
        } else {
            /// If no, calculate the lockupEndTime
            lockupEndTime = endTime +
fixedPriceSaleConfig.lockupPeriodSeconds;
        }
        // Set the vestingStartTime to begin when lockupEndTime is
reached
        vestingStartTime = lockupEndTime;
        /// Verify if the sale configuration is valid
        _verifyValidConfig(fixedPriceSaleConfig);
>>
        /// Cache Legion addresses from `LegionAddressRegistry`
        legionBouncer =
ILegionAddressRegistry(addressRegistry).getLegionAddress(LEGION_BO
UNCER_ID);
        legionSigner =
ILegionAddressRegistry(addressRegistry).getLegionAddress(LEGION_SI
GNER_ID);
        legionFeeReceiver =
ILegionAddressRegistry(addressRegistry).getLegionAddress(LEGION_FE
E_RECEIVER_ID);
        vestingFactory =
ILegionAddressRegistry(addressRegistry).getLegionAddress(LEGION_VE
STING_FACTORY_ID);
    }
```

Although the owner can assign a new address for a given ID in LegionAddressRegistry.sol, the sale will still use the old address:

```
function setLegionAddress(bytes32 id, address updatedAddress)
external onlyOwner {
    /// Cache the previous address before update
    address previousAddress = _legionAddresses[id];
    /// Update the address in the state
    _legionAddresses[id] = updatedAddress;
    /// Successfully emit LegionAddressSet
```

```
emit LegionAddressSet(id, previousAddress,
updatedAddress);
}
```

For instance, if LEGION_SIGNER_ID is compromised after the sale deployment, updating it with setLegionAddress(LEGION_SIGNER_ID, newSigner) won't affect the sale, which will continue using the compromised address.

Recommendation: It is recommended to implement an additional function that allows to sync sale addresses in the sale contract with addresses from the AddressRegistry.sol:

```
function syncAddresses() external onlyLegion {
    legionBouncer =
ILegionAddressRegistry(addressRegistry).getLegionAddress(LEGION_BO
UNCER_ID);
    legionSigner =
ILegionAddressRegistry(addressRegistry).getLegionAddress(LEGION_SI
GNER_ID);
    legionFeeReceiver =
ILegionAddressRegistry(addressRegistry).getLegionAddress(LEGION_FE
E_RECEIVER_ID);
    vestingFactory =
ILegionAddressRegistry(addressRegistry).getLegionAddress(LEGION_VE
STING_FACTORY_ID);
}
```

Legion:

The issue has been fixed with PR-2

C4 Zenith:

The issue has been resolved as per recommendation.



9. Informational Findings

A total of 2 informational findings were identified.

9.1. Misplaced values in saleConfiguration() function

Sev	erity:	Inform	national	
-----	--------	--------	----------	--

Status: Resolved

Context:

- LegionFixedPriceSale.sol#L178-L179
- ILegionFixedPriceSale.sol#L42-L73

Description:

The saleConfiguration() function returns a FixedPriceSaleConfig structure with various sale configuration parameters:

```
function saleConfiguration() external view returns
(FixedPriceSaleConfig memory saleConfig) {
        /// Get the fixed price sale config
        saleConfig = FixedPriceSaleConfig(
            prefundPeriodSeconds,
            prefundAllocationPeriodSeconds,
            salePeriodSeconds,
            refundPeriodSeconds,
            lockupPeriodSeconds,
            vestingDurationSeconds,
            vestingCliffDurationSeconds,
            legionFeeOnCapitalRaisedBps,
            legionFeeOnTokensSoldBps,
            tokenPrice,
>>
>>
            minimumPledgeAmount,
            bidToken,
            askToken,
            projectAdmin,
            addressRegistry
        );
    }
```



Upon review of the ILegionFixedPriceSale.sol interface, it appears that the tokenPrice and minimumPledgeAmount parameters are misplaced and should be swapped to match their correct positioning.

Recommendation:

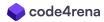
Correct the parameter order by swapping tokenPrice and minimumPledgeAmount.

Legion:

The issue has been fixed with the following commit

C4 Zenith:

The issue has been resolved as per recommendation.



9.2. ECIES#isValid() Validity check is not complete

Severity: Informational

Status: Resolved

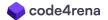
Context:

• ECIES.sol#L138

Description:

isValid() and decrypt() current implementation:

```
function isValid(Point memory p) public pure returns (bool) {
        return isOnBn128(p) && !(p.x == 1 && p.y == 2) && !(p.x ==
Ø && p.y == ∅);
    }
// ECIES.sol
function decrypt(
        uint256 ciphertext_,
        Point memory ciphertextPubKey_,
        uint256 privateKey_,
        uint256 salt_
    ) public view returns (uint256 message_) {
        // Calculate the shared secret
        // Validates the ciphertext public key is on the curve and
the private key is valid
        uint256 sharedSecret =
recoverSharedSecret(ciphertextPubKey_, privateKey_);
        . . .
    }
  function recoverSharedSecret(
        Point memory ciphertextPubKey_,
        uint256 privateKey_
    ) public view returns (uint256) {
              . . .
        Point memory p = _ecMul(ciphertextPubKey_, privateKey_);
```



```
return p.x;
}
function _ecMul(Point memory p, uint256 scalar) private view
returns (Point memory p2) {
    (bool success, bytes memory output) =
        address(0x07).staticcall{gas: 60000}(abi.encode(p.x,
p.y, scalar));
    if (!success || output.length == 0) revert("ecMul
failed.");
    p2 = abi.decode(output, (Point));
}
```

isValid() Validity check is not complete, May cause decrypt() to failure

Among other things, recoverSharedSecret() will execute a scalar multiplication between the invalid public key and the global private key via the ecMul precompile. This is where the denial of servide will take place.

more detailed description

The pubKey is provided by legion so There are currently no security risks. However, since this is a tool function, it is recommended add check to avoid subsequent use elsewhere.

Recommendation:

```
function isValid(Point memory p) public pure returns (bool) {
    return isOnBn128(p) && !(p.x == 1 && p.y == 2) && !(p.x ==
    && p.y == 0);
    return isOnBn128(p) && !(p.x == 1 && p.y == 2) && !(p.x ==
    && p.y == 0)&& (p.x < FIELD_MODULUS) && (p.y < FIELD_MODULUS)
    }
}</pre>
```

Legion:

The issue has been fixed with the following commit



C4 Zenith:

The issue has been resolved as per recommendation

