Code Security Audit Report

For

Substancex

23th Jan 2025

Table of Contents

Summary

- **Project Executive Summary**
- **Vulnerability Summary**
- **Audit Scope Summary**
- **Audit Approach Summary**

Audit Result

- ✓ **SSX-01(Critical): Neglecting to Convert USD to Margin Token Leads to Incorrect Accounting and Functional Disruptions.**
- ✓ **SSX-02(High): USD Stablecoins are Incorrectly Assumed to Always be at peg.**
- ✓ **SSX-03(High): Incorrect Price Restriction for Market Orders.**
- ✓ **SSX-04(High): Incorrect Leverage Validation When Increasing Position.**
- ✓ **SSX-05(High): Incorrect Leverage Check in makeIncreaseLimitOrder.**
- ✓ **SSX-06(Medium): Single-step Ownership Transfer Can be Dangerous.**
- ✓ **SSX-07(Medium): No Storage Gap for Upgradeable Contract Might Lead to Storage Slot Collision.**
- ✓ **SSX-08(Medium): Chainlink's latestRoundData might return stale**

or incorrect results.

- ✓ **SSX-09(Low): Missing Order Parameter Validation in makeIncreaseMarketOrder.**
- ✓ **SSX-10(Low): Missing Margin Price Validation.**
- ✓ **SSX-11(Info): Unused Storage Variables.**
- ✓ **SSX-12(Info): Floating Pragma Should Be Avoided.**
- ✓ **SSX-13(Info): Missing Event for increaseCollateral.**

Appendix

About

Damocles

Project Executive Summary

TYPES	MAINNET	AUDIT METHOD
FUTURE	Ethereum Chain	Cross-manual review; Static Analysis
DEVELOP LAUNGE	**CODEBASE**	
SOLIDITY	Contract Link	**Commit ID** 2f1eded7d66
FIXED CODEBASE	Contract Link	120b221ea87

Vulnerability Summary

13	2	0	11
Total Findings	Solved	Pending	Ack

SEVERITY	STATUS	DESCRIPTION
CRITICAL	**1 Solved**	Critical risks are those that impact the safe functioning of a platform and must be addressed before launch. Users should not invest in any project with outstanding critical risk
HIGH	**1 Solved** 3 Acknowledged	High risks can include centralization issues and logical errors. Under specific circumstances, these high risks can lead to loss of funds and/or control of the project
MEDIUM	**3 Acknowledged**	Medium risks may not pose a direct risk to user's funds, but they can affect the overall functioning of a platform
LOW	**2 Acknowledged**	Low risks can be any of the above, but on a smaller scale. They generally do not compromise the over all integrity of the project, but they may be less efficient than other solution

INFO	**3 Acknowledged**	Info errors are often recommendations to improve the style of the code or certain operations to fall within industry best practices. They usually do not affect the overall functioning of the code

Audit Scope Summary

Code Repository: https://github.com/SubstanceExchangeDevelop/contract/tree/feature/zeta	
File Name	**Commit ID**
CryptoFuturesLong.sol	2f1eded7d66
CryptoFuturesManager.sol	2f1eded7d66
CryptoFuturesShort.sol	2f1eded7d66
CryptoLiquidityPool.sol	2f1eded7d66
UserBalanceV2.sol	2f1eded7d66
CryptoLimitOrder.sol	2f1eded7d66
CryptoMarketOrder.sol	2f1eded7d66
CryptoOrder.sol	2f1eded7d66
CryptoStopOrder.sol	2f1eded7d66
CryptoUpdateCollateralOrder.sol	2f1eded7d66

Damocles

Audit Approach Summary

This report has been prepared **Substancex** to discover issues and vulnerabilities in the source code of the **Substancex Future Repo** as well as any contract dependencies that were not part of an officially recognized library. A comprehensive examination has been performed, utilizing Cross Manual Review and Static Analysis techniques

- The auditing process pays special attention to the following considerations:
- Testing the smart contracts against both common and uncommon attack vectors.
- Assessing the codebase to ensure compliance with current best practices and industry standards.
- Ensuring contract logic meets the specifications and intentions of the client.
- Cross referencing contract structure and implementation against similar smart contracts produced by industry leaders
- .Thorough line-by-line manual review of the entire codebase by industry experts
- Cross-audit mode of the current code by more than three security engineers.

The security assessment resulted in findings that ranged from critical to informational. We recommend addressing these findings to ensure a high level of security standards and industry practices. We suggest recommendations that could better serve the project from the security perspective

- Testing the smart contracts against both common and uncommon attack vectors;
- Enhance general coding practices for better structures of source codes;
- Add enough unit tests to cover the possible use cases;
- Provide more comments per each function for readability, especially contracts that are verified in public;
- Provide more transparency on privileged activities once the protocol is live

Damocles

Audit Result

SSX-01(Critical): Improper configuration leading to the airdrop logic not being executed correctly

Category	Severity	Location	Status
Code Issue	Critical	CryptoBaseFutures.sol:562	Solved

Description

The protocol uses non-stablecoins as collateral, but due to oversight, USD is not consistently converted into margin tokens in multiple instances. This results in incorrect accounting and functional disruptions.

Vulnerability Analysis

Examples of the Issue:

1. In the **getMaxDecreaseCollateral** Function

The function calculates **minRemainCollateral** using **remainCollateralRatio**. However, the result is not converted to the corresponding margin token amount, leading to accounting inaccuracies.

```
uint256 remainCollateralUSD = (positionValueUSD * (reaminCollateralRatio[_futureId] + _predictedFeeRatio)) / 10000;
```

In contrast, the **checkLiquidation** function demonstrates the

correct handling by converting the value into margin tokens:

```
uint256 remainCollateralUSD = (positionValueUSD * (reaminCollateralRatio[_futureId] + _predictedFeeRatio)) / 10000;

uint256 netValueToken = convertUSDToMargin(SafeCast.toUint256(netValueUSD > 0 ? netValueUSD : -netValueUSD), _marginPrice);
uint256 remainCollateralToken = convertUSDToMargin(remainCollateralUSD, _marginPrice);
```

2. In the **checkMaxProfitClosePosition** Function

The decreaseFees calculation does not convert the result into the corresponding margin token amount:

```
  uint256 decreaseFees = (positionValue * _predictedFeeRatio) / 10000;
  _executePositionDecrease(_futureId, position, position.tokenSize, _price, _marginPrice, decreaseFees, _availableLp, result);
```

A correct implementation can be found in the **decreasePosition** function:

```
uint256 txFeeUSD = _chargeTransactionFee(_futureId, _decreaseTokenSize, _price, _feeInfo.txFeeRatio);
uint256 piFeeUSD = _chargePriceImpactFee(_futureId, _decreaseTokenSize, _price, _feeInfo.priceImpactRatio);
uint256 txFeeMargin = convertUSDToMargin(txFeeUSD, _marginPrice);
uint256 piFeeMargin = convertUSDToMargin(piFeeUSD, _marginPrice);
_executePositionDecrease(_futureId, position, _decreaseTokenSize, _price, _marginPrice, txFeeMargin + piFeeMargin, _feeInfo.availableLp, result);
```

3. In the **getNetValue** Function

While **netValueInToken** and **pnlInToken** are converted into margin token values, **fundingFeeInToken** and **borrowingFeeInToken** are not converted. However, all these values are used together in calculating **netValueInToken**, leading to inaccuracies:

```
function getNetValue(
    uint256 _futureId,
    Struct.Position memory position,
    uint256 _price,
    uint256 _marginPrice
) public view returns (int256 netValueInToken) {
    netValueInToken = SafeCast.toInt256(position.collateral);

    (int256 pnlInToken, ) = getPositionPnl(_futureId, position, _price, _marginPrice);

    int256 fundingFeeInToken = calcFundingFee(_futureId, position.tokenSize, position.entryFundingFeePerToken, position.cumulativeFundingFee);

    uint256 borrowingFeeInToken = calcBorrowingFee(_futureId, position.tokenSize, position.entryBorrowingFeePerToken, position.cumulativeBorrowingFee);
    uint256 txFeeInToken = position.cumulativeTeamFee;

    netValueInToken = netValueInToken + pnlInToken - fundingFeeInToken - SafeCast.toInt256(borrowingFeeInToken) - SafeCast.toInt256(txFeeInToken);
}
```

Note: The protocol contains similar issues in several locations; the above are just examples.

Recommendation

Adjust the code logic to ensure USD values are consistently converted into margin tokens wherever applicable.

SSX-02(High): USD Stablecoins are Incorrectly Assumed to Always be at peg.

Category	Severity	Location	Status
Code Issue	High	CryptoFuturesLong.sol:23 45	Acknowledged

Description

The protocol's assumption that stablecoins are equal to 1 USD may lead to economic losses.

Vulnerability Analysis

The protocol allows stablecoins (e.g., USDC, USDT) to be used as collateral and assumes their value is always perfectly pegged to 1 USD. However, during extreme market conditions, stablecoins may lose their peg, causing their actual value to drop below 1 USD. For instance, USDC temporarily depegged to 0.87 USD during the Silicon Valley Bank crisis in March 2023.

This assumption introduces significant risks:

1. When the value of stablecoin collateral drops, the protocol may overestimate the collateral value, delaying necessary liquidations and increasing protocol exposure.
2. Attackers may exploit depegging by depositing stablecoins worth less than 1 USD as collateral, gaining benefits beyond their fair value.
3. Depegged stablecoin collateral could lead to under-collateralized positions, negatively affecting the protocol's liquidity pool and debt recovery process.

Recommendation

It is recommended to use decentralized oracles to fetch real-time market prices of stablecoins and calculate collateral value based on actual prices rather than assuming a fixed 1 USD value.

SSX-03(High): Incorrect Price Restriction for Market Orders.

Category	Severity	Location	Status
Code Issue	High	CryptoMarketOrder.sol:107 121	Acknowledged

Description

Incorrect leverage limit when executing Increase Market Order may cause the order to fail.

Vulnerability Analysis

In the **_executeIncreaseMarketOrder** and **_executeDecreaseMarketOrder** functions, market orders are executed with the following checks:

1. A deadline check ensures the order is not expired.

2. A price check, similar to limit order validation, cancels the order if:

- For increasing a long position, the **currentPrice** exceeds the order's **executePrice**.
- For decreasing a long position, the **currentPrice** is below the order's **executePrice**.

This price check undermines the primary purpose of market orders, making them functionally similar to limit orders and negating their intended advantage of rapid execution at the current market price.

```
if (order.future == manager.futureLong()) {
    if (_currentPrice > order.executePrice) {
        cancelled = true;
    }
} else {
    if (_currentPrice < order.executePrice) {
        cancelled = true;
    }
}
```

Consider a scenario where the price of a token is steadily increasing, and a user places a market order to go long to profit from the uptrend. The user deliberately avoids a limit order to ensure immediate execution. However, due to the continuous price increase, the price check fails, and the order is canceled. This results in missed trading opportunities and contradicts the intended use of market orders.

Recommendation

To improve the execution of market orders, choose one of the following approaches based on the protocol's requirements:

1. Market orders are meant to execute at the current price without restrictions. Removing the price check ensures the market order fulfills its primary purpose of immediate execution, regardless of

price fluctuations.

2. To maintain a safeguard against extreme market fluctuations, introduce a tolerance range (e.g., within 1% of the executePrice). This allows small deviations while still prioritizing execution speed.

SSX-04(High):Incorrect Leverage Validation When Increasing Position.

Category	Severity	Location	Status
Logic Issue	High	CryptoLimitOrder.sol:111 173	Acknowledged

Description

When executing Increase Market Order, the original position collateral and position were not considered.

Vulnerability Analysis

In the process of increasing a position, the following leverage checks are performed:

Damocles

```
uint256 usdValue = ICryptoFutures(future).getUSDValue(futureId, increaseTokenSize, _curPrice);
if (usdValue < _getConfig(ConfigConstants.ORDER_MIN_USD_VALUE, futureId)) {
    _cancelIncreaseLimitOrderWithFund(_user, _nonce, Struct.CancelReason.LessThanMinUSDValue, _feeInfo.feeReceiver);
    return;
}
uint256 collateralUSD = ICryptoFutures(future).convertMarginToUSD(increaseCollateral, _curMarginPrice);
if (usdValue < (_getConfig(ConfigConstants.ORDER_MIN_LEVERAGE, futureId) * collateralUSD) / ConfigConstants.LEVERAGE_PRECISION) {
    _cancelIncreaseLimitOrderWithFund(_user, _nonce, Struct.CancelReason.LessThanMinLeverage, _feeInfo.feeReceiver);
    return;
}
if (usdValue > (_getConfig(ConfigConstants.ORDER_MAX_LEVERAGE, futureId) * collateralUSD) / ConfigConstants.LEVERAGE_PRECISION) {
    _cancelIncreaseLimitOrderWithFund(_user, _nonce, Struct.CancelReason.MoreThanMaxLeverage, _feeInfo.feeReceiver);
    return;
}
```

These checks only consider the **increaseCollateral** and **increaseTokenSize** of the current order to validate leverage limits. However, this approach ignores the existing position's **TokenSize** and **Collateral**.

Assume the protocol defines the allowable leverage range as 1–5. A user currently has a position with:

- TokenSize = 100

- Collateral = 50

- Current leverage = 2

If the user tries to increase the position by:

- increaseTokenSize = 10

- increaseCollateral = 1

The leverage for this increment alone would be **10 / 1 = 10**, exceeding the maximum allowable leverage. However, after the increase:

- New TokenSize = 110

- New Collateral = 51

- New leverage = **110 / 51 ≈ 2.16**, which is within the acceptable

range of 1–5.

The current logic incorrectly cancels this valid order due to its isolated leverage check on the increment, leading to failure in processing legitimate user orders.

Recommendation

Update the logic to calculate the leverage based on the combined position (existing and incremental).

SSX-05(High):Incorrect Leverage Check in makeIncreaseLimitOrder.

Category	Severity	Location	Status
Logic Issue	High	CryptoLimitOrder.sol:77 109	Solved

Description

Assuming collateral is stablecoin when making Increase Limit Order may cause most user orders to fail.

Vulnerability Analysis

In the **makeIncreaseLimitOrder** function, the protocol performs the following leverage checks:

```
uint256 usdValue = ICryptoFutures(_future).getUSDValue(_futureId, _increaseTokenSize, _price);
if (usdValue < _getConfig(ConfigConstants.ORDER_MIN_USD_VALUE, _futureId)) revert LimitOrder__LessThanMinUSDValue();
if (usdValue < (_getConfig(ConfigConstants.ORDER_MIN_LEVERAGE, _futureId) * _increaseCollateral) / ConfigConstants.LEVERAGE_PRECISION)
    revert LimitOrder__LessThanMinLeverage();
if (usdValue > (_getConfig(ConfigConstants.ORDER_MAX_LEVERAGE, _futureId) * _increaseCollateral) / ConfigConstants.LEVERAGE_PRECISION)
    revert LimitOrder__MoreThanMaxLeverage();
```

This approach assumes **_increaseCollateral** represents a stablecoin value and directly compares it with **usdValue** for leverage calculations. However, since the margin token is not a stablecoin, this incorrect assumption causes most orders to fail leverage validation, leading to failed order creation.

Example:

- Leverage range: 1-5.

- Collateral: 1 A token worth $1,000.

- Order: $2,000 position with 1 A token as collateral (2x leverage).

- Due to incorrect leverage calculation, the protocol calculates leverage as 2,000x instead of 2x, causing the order to fail.

Recommendation

To fix this issue, calculate the value of the margin token in USD and use it for leverage validation, as done in the **executeIncreaseLimitOrder** function:

Damocles

```
uint256 usdValue = ICryptoFutures(future).getUSDValue(futureId, increaseTokenSize, _curPrice);
if (usdValue < _getConfig(ConfigConstants.ORDER_MIN_USD_VALUE, futureId)) {
    _cancelIncreaseLimitOrderWithFund(_user, _nonce, Struct.CancelReason.LessThanMinUSDValue, _feeInfo.feeReceiver);
    return;
}
uint256 collateralUSD = ICryptoFutures(future).convertMarginToUSD(increaseCollateral, _curMarginPrice);
if (usdValue < (_getConfig(ConfigConstants.ORDER_MIN_LEVERAGE, futureId) * collateralUSD) / ConfigConstants.LEVERAGE_PRECISION) {
    _cancelIncreaseLimitOrderWithFund(_user, _nonce, Struct.CancelReason.LessThanMinLeverage, _feeInfo.feeReceiver);
    return;
}
if (usdValue > (_getConfig(ConfigConstants.ORDER_MAX_LEVERAGE, futureId) * collateralUSD) / ConfigConstants.LEVERAGE_PRECISION) {
    _cancelIncreaseLimitOrderWithFund(_user, _nonce, Struct.CancelReason.MoreThanMaxLeverage, _feeInfo.feeReceiver);
    return;
}
```

SSX-06(Medium): Single-step Ownership Transfer Can be Dangerous.

Category	Severity	Location	Status
Logic Issue	Medium	CryptoFuturesManager.sol:10	Acknowledged

Description

Single-step ownership transfer means that if a wrong address was passed when transferring ownership or admin rights it can mean that role is lost forever.

Vulnerability Analysis

Single-step ownership transfer means that if a wrong address was passed when transferring ownership or admin rights it can mean that role is lost forever. If the admin permissions are given to the wrong address within this function, it will cause irreparable damage to the contract.

Below is the official documentation explanation from OpenZeppelin:

https://docs.openzeppelin.com/contracts/4.x/api/access

Ownable is a simpler mechanism with a single owner "role" that can be assigned to a single account. This simpler mechanism can be useful for quick tests but projects with production concerns are likely to outgrow it.

Recommendation

It is recommended to use a two-step ownership transfer pattern, meaning ownership transfer gets to a "pending" state and the new owner should claim his new rights, otherwise the old owner still has control of the contract.

SSX-07(Medium): No Storage Gap for Upgradeable Contract Might Lead to Storage Slot Collision.

Category	Severity	Location	Status
Logic Issue	Medium	CryptoBaseFutures.sol	Acknowledged

Description

For upgradeable contracts, the absence of a storage gap may lead to storage conflicts during future upgrades.

Vulnerability Analysis

For upgradeable contracts, there must be storage gap to allow developers to freely add new state variables in the future without compromising the storage compatibility with existing deployments (quote OpenZeppelin). Otherwise it may be very difficult to write new implementation code. Without storage gap, the variable in child contract might be overwritten by the upgraded base contract if new variables are added to the base contract. This could have unintended and very serious consequences to the child contracts, potentially causing loss of user fund or cause the contract to malfunction completely. The FutureLong and FutureShort contracts inherit from the upgradeable contract BaseFuture. The BaseFuture contract has no storage gaps, which could lead to potential storage collision issues in future upgrades. For more information, please refer to: https://docs.openzeppelin.com/upgrades-plugins/1.x/writing-upgradeable#storage-gaps.

Recommendation

Recommend adding appropriate storage gap at the end of upgradeable contracts such as the below. Please reference OpenZeppelin upgradeable contract templates.

```
uint256[50] private __gap;
```

SSX-08(Medium): Chainlink's latestRoundData might

return stale or incorrect results.

Category	Severity	Location	Status
Logic Issue	Medium	PriceOracle.sol:58 67	Acknowledged

Description

The Chainlink oracle may return outdated prices, and the protocol does not perform any checks for this.

Vulnerability Analysis

In the **PriceOracle** contract, the protocol uses a Chainlink aggregator to fetch the **latestRoundData()**. However, there is no verification to determine if the returned data is stale. The only existing validation checks whether the **quoteAnswer (oraclePrice)** is greater than 0, but this is insufficient. There is a risk that the data may be outdated or incorrect due to issues with the Chainlink oracle, such as delayed updates or failures in the data feed. Without checking for staleness, the protocol is exposed to the risk of relying on invalid price data.

Damocles

```
function validateSubproductPrice(bytes32 product, uint256 price) internal view returns (bool, int256) {
    // Substance Oracle Price is always 6 decimals
    OracleInfo memory info = oracle[product];
    if (info.oracle != address(0)) {
        uint8 oracleDecimals = AggregatorV3Interface(info.oracle).decimals();
        (, int256 oraclePrice, , , ) = AggregatorV3Interface(info.oracle).latestRoundData();
        if (oraclePrice <= 0) revert PriceOracle__InvalidPrice(oraclePrice);
        price = (price * (10**oracleDecimals)) / (10**(6 + info.decimalOffset));
        int256 acceptanceRange = oraclePrice * SafeCast.toInt256(SafeCast.toUint64(info.acceptanceThreshold));
        int256 diff = (oraclePrice - SafeCast.toInt256(price)) * SafeCast.toInt256(ACCEPTANCE_PRECISION);
        if ((diff < 0 && (-diff > acceptanceRange)) || (diff >= 0 && (diff > acceptanceRange))) {
            return (false, oraclePrice);
        }
    }
    return (true, 0);
}
```

Recommendation

Add checks to validate the freshness and completeness of the price data, such as:

```
(uint80 roundID, int256 price, , uint256 timestamp, uint80 answeredInRound) =  AggregatorV3Interface(info.oracle).latestRoundData();
require(answeredInRound >= roundID, "Stale price!");
require(timestamp != 0, "Incomplete round!");
require(block.timestamp - timestamp <= VALID_TIME_PERIOD, "Price too old!");
require(price > 0, "Invalid price!");
```

SSX-09(Low): Missing Order Parameter Validation in makeIncreaseMarketOrder.

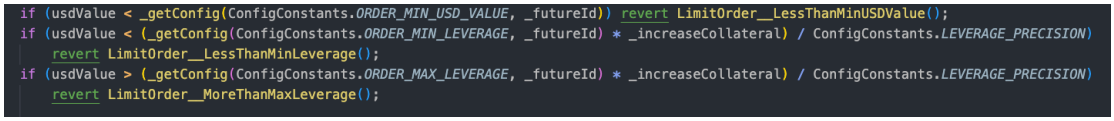
Category	Severity	Location	Status
Logic Issue	Low	CryptoMarketOrder.sol:77 105	Acknowledged

Description

The **makeIncreaseMarketOrder** function lacks the same parameter validation checks present in the **makeIncreaseLimitOrder** function.

Vulnerability Analysis

In **makeIncreaseLimitOrder**, the following validations are performed:



```
if (usdValue < _getConfig(ConfigConstants.ORDER_MIN_USD_VALUE, _futureId)) revert LimitOrder__LessThanMinUSDValue();
if (usdValue < (_getConfig(ConfigConstants.ORDER_MIN_LEVERAGE, _futureId) * _increaseCollateral) / ConfigConstants.LEVERAGE_PRECISION)
    revert LimitOrder__LessThanMinLeverage();
if (usdValue > (_getConfig(ConfigConstants.ORDER_MAX_LEVERAGE, _futureId) * _increaseCollateral) / ConfigConstants.LEVERAGE_PRECISION)
    revert LimitOrder__MoreThanMaxLeverage();
```

However, the **makeIncreaseMarketOrder** function lacks this check.

Recommendation

To maintain consistency, include the same parameter validation in **makeIncreaseMarketOrder**.

SSX-10(Low): Missing Margin Price Validation.

Category	Severity	Location	Status
Logic Issue	Low	CryptoFuturesManager.sol:225 243	Acknowledged

Description

The protocol only verifies the price of the position token, lacking validation for the margin token's price.

Vulnerability Analysis

The current protocol validates the future price during order execution, collateral adjustments, and liquidation by calling:

Damocles

```
function liquidatePosition(
    uint256 _futureId,
    address _user,
    uint256 _curPrice,
    uint256 _marginPrice,
    Struct.OrderFeeInfo calldata _feeInfo,
    Struct.FutureType _futureType
) external onlyOperator {
    address _future = getFutureByType(_futureType);
    oracle.validatePrice(_future, _futureId, _curPrice);
    Struct.UpdatePositionResult memory result = ICryptoFutures(_future).liquidatePosition(
        _user,
        _futureId,
        _curPrice,
        _marginPrice,
        getFeeInfoWithAvailableToken(_user, false, _feeInfo)
    );
    _settleFuture(_future, _futureId, _user, result);
}
```

However, no validation is performed for the margin price, which may lead to inaccuracies in the execution price. This lack of validation could also be exploited by a malicious operator, posing a risk to the protocol's security.

Recommendation

Add a validation step for the margin price to ensure accuracy and prevent potential exploitation by operators.

SSX-11(Info): Unused Storage Variables.

Category	Severity	Location	Status
Logic Issue	Info	CryptoBaseFutures.sol:41 42	Acknowledged

Description

Unused storage variables can lead to additional gas consumption

and may also indicate incomplete functionality implementation.

Vulnerability Analysis

The following storage variables are declared but not used in the contract:

```
mapping(uint256 => uint256) public currentEpoch;
mapping(uint256 => uint256) public currentEpochEndTime;
```

Unused storage variables increase the deployment cost of the contract.

Recommendation

If these variables are redundant or will not be used, remove them from the contract to reduce deployment costs and simplify maintenance.

SSX-12(Info): Floating Pragma Should Be Avoided.

Category	Severity	Location	Status
Logic Issue	Info	CryptoBaseFutures.sol:8	Acknowledged

Description

Using a floating Solidity compiler version may lead to unexpected issues and security risks.

Vulnerability Analysis

The use of **pragma solidity ^0.8.19** specifies a floating version of the Solidity compiler. This means the contract is compatible with version **0.8.19** or any higher version below **0.9.0**. While this provides some flexibility, it introduces potential risks:

1. Future compiler versions may introduce breaking changes or subtle behavioral differences, which could unintentionally affect the functionality or security of the contract.
2. It reduces determinism for testing and deployment since the exact compiler version used cannot be guaranteed across different environments.
3. Starting from Solidity **0.8.20**, the compiler targets the new Shanghai EVM, which introduces the **PUSH0** opcode. This opcode may not be implemented on all chains or Layer 2 solutions, leading to potential incompatibility or failures during execution.

Recommendation

Use a fixed compiler version, such as **pragma solidity 0.8.19**.

Damocles

SSX-13(Info): Missing Event for increaseCollateral.

Category	Severity	Location	Status
Logic Issue	Info	BaseFuture.sol:338 383	Acknowledged

Description

The lack of events in the current protocol can lead to difficulties in tracking.

Vulnerability Analysis

In the **decreaseCollateral** function, the following event is emitted:

```
event UpdateCollateral(address indexed user, uint256 indexed futureId, bool increase, uint256 marginAmount, uint256 marginPrice);
```

However, the **increaseCollateral** function does not emit this event.

Recommendation

Add the event in the **increaseCollateral** function.

APPENDIX

Audit Categories

Categories	Description
Code Issues	Coding Issue findings are about general code quality including, but not limited to, coding mistakes, compile errors, and performance issues
Volatile Code	Volatile Code findings refer to segments of code that behave unexpectedly on certain edge cases and may result in vulnerabilities
Logic Issues	Logic Issue findings indicate general implementation issues related to the program logic.
Centralization	Centralization findings detail the design choices of designating privileged roles or other centralizedcontrols over the code

About

Damocles is a 2023 web3 security company specializing in online security services, including smart contract audit, Product audit, penetration testing, GameFi security audit and cheat detection.

Main Web: https://damocleslabs.com/

Twitter: https://twitter.com/DamoclesLabs

Email: support@damocleslabs.com