Increment Finance: Increment Protocol

Fix Review

November 10, 2022

Prepared for:
Increment Finance

Prepared by: **Anish Naik, Justin Jacob, and Vara Prasad Bandaru**

About Trail of Bits

Founded in 2012 and headquartered in New York, Trail of Bits provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code. With 100+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

We maintain an exhaustive list of publications at https://github.com/trailofbits/publications, with links to papers, presentations, public audit reports, and podcast appearances.

In recent years, Trail of Bits consultants have showcased cutting-edge research through presentations at CanSecWest, HCSS, Devcon, Empire Hacking, GrrCon, LangSec, NorthSec, the O'Reilly Security Conference, PyCon, REcon, Security BSides, and SummerCon.

We specialize in software testing and code review projects, supporting client organizations in the technology, defense, and finance industries, as well as government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, and Zoom.

Trail of Bits also operates a center of excellence with regard to blockchain security. Notable projects include audits of Algorand, Bitcoin SV, Chainlink, Compound, Ethereum 2.0, MakerDAO, Matic, Uniswap, Web3, and Zcash.

To keep up to date with our latest news and announcements, please follow @trailofbits on Twitter and explore our public repositories at https://github.com/trailofbits. To engage us directly, visit our "Contact" page at https://www.trailofbits.com/contact, or email us at info@trailofbits.com.

Trail of Bits, Inc.
228 Park Ave S #80688
New York, NY 10003
https://www.trailofbits.com
info@trailofbits.com

Notices and Remarks

Copyright and Distribution

Test Coverage Disclaimer

All activities undertaken by Trail of Bits in association with this project were performed in accordance with a statement of work and agreed upon project plan.

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

Trail of Bits uses automated testing techniques to rapidly test the controls and security properties of software. These techniques augment our manual security review work, but each has its limitations: for example, a tool may not generate a random edge case that violates a property or may not fully complete its analysis during the allotted time. Their use is also limited by the time and resource constraints of a project.

When undertaking a fix review, Trail of Bits reviews the fixes implemented for issues identified in the original report. This work involves a review of specific areas of the source code and system configuration, not comprehensive analysis of the system.

Table of Contents

Executive Summary

Engagement Overview

Increment Finance engaged Trail of Bits to review the security of its Increment Protocol. From August 22 to September 2, 2022, a team of two consultants conducted a security review of the client-provided source code, with four person-weeks of effort. Details of the project's scope, timeline, test targets, and coverage are provided in the original audit report.

Increment Finance contracted Trail of Bits to review the fixes implemented for issues identified in the original report. On October 25, 2022, one consultant conducted a review of the client-provided source code, with four person-hours of effort.

Summary of Findings

The original audit uncovered significant flaws that could impact system confidentiality, integrity, or availability. A summary of the original findings is provided below.

EXPOSURE ANALYSIS

Severity	*Count*
High	3
Medium	3
Low	2
Informational	5
Undetermined	0

CATEGORY BREAKDOWN

Category	*Count*
Access Controls	1
Configuration	1
Data Validation	5
Patching	1
Timing	2
Undefined Behavior	3

Overview of Fix Review Results

Increment Finance has sufficiently addressed most of the issues described in the original audit report.

Project Summary

Contact Information

The following managers were associated with this project:

Dan Guido, Account Manager
dan@trailofbits.com

Anne Marie Barry, Project Manager
annemarie.barry@trailofbits.com

The following engineers were associated with this project:

Anish Naik, Consultant
anish.naik@trailofbits.com

Justin Jacob, Consultant
justin.jacob@trailofbits.com

Vara Prasad Bandaru, Consultant
vara.bandaru@trailofbits.com

Project Timeline

The significant events and milestones of the project are listed below.

Date	Event
August 18, 2022	Pre-project kickoff call
August 26, 2022	Status update meeting #1
September 2, 2022	Report readout meeting and delivery of report draft
November 10, 2022	Delivery of final report and fix review

Project Methodology

Our work in the fix review included the following:

- A review of the findings in the original audit report
- A manual review of the client-provided source code and configuration material
- A review of the documentation provided alongside the codebase

Project Targets

The engagement involved a review of the fixes implemented in the following target.

Increment Protocol

Repository	https://github.com/Increment-Finance/increment-protocol
Version	`9368b23ac2d2f5dc954cc849d20cdeca21d627c6`
Type	Solidity
Platform	zkSync

Summary of Fix Review Results

The table below summarizes each of the original findings and indicates whether the issue has been sufficiently resolved.

ID	Title	Status
1	Governance role is a single point of failure	**Partially Resolved**
2	Inconsistent lower bounds on collateral weights	**Resolved**
3	Solidity compiler optimizations can be problematic	**Unresolved**
4	Support for multiple reserve tokens allows for arbitrage	**Unresolved**
5	Ownership transfers can be front-run	**Resolved**
6	Funding payments are made in the wrong token	**Resolved**
7	Excessive dust collection may lead to premature closures of long positions	**Resolved**
8	Problematic use of primitive operations on fixed-point integers	**Resolved**
9	Liquidations are vulnerable to sandwich attacks	**Resolved**
10	Accuracy of market and oracle TWAPs is tied to the frequency of user interactions	**Resolved**
11	Liquidations of short positions may fail because of insufficient dust collection	**Resolved**
12	Project dependencies contain vulnerabilities	**Resolved**

13	Risks associated with oracle outages	Unresolved

Detailed Fix Review Results

1. Governance role is a single point of failure	
Status: **Partially Resolved**	
Severity: **High**	Difficulty: **High**
Type: Access Controls	Finding ID: TOB-INC-1
Target: Governance role	

Description

Because the governance role is centralized and responsible for critical functionalities, it constitutes a single point of failure within the Increment Protocol.

The role can perform the following privileged operations:

- Whitelisting a perpetual market
- Setting economic parameters
- Updating price oracle addresses and setting fixed prices for assets
- Managing protocol insurance funds
- Updating the addresses of core contracts
- Adding support for new reserve tokens to the UA contract
- Pausing and unpausing protocol operations

These privileges give governance complete control over the protocol and therefore access to user and protocol funds. This increases the likelihood that the governance account will be targeted by an attacker and incentivizes governance to act maliciously.

Note, though, that the governance role is currently controlled by a multisignature wallet (a multisig) and that control may be transferred to a decentralized autonomous organization (DAO) in the future.

Fix Analysis
This issue has been partially resolved. The Increment Finance team minimized the privileges of the governance role by removing its ability to update core contract addresses (i.e., allowing it to set those addresses only once). No other privileges were removed.

2. Inconsistent lower bounds on collateral weights

Status: **Resolved**	
Severity: **Medium**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-INC-2
Target: `contracts/Vault.sol`	

Description

The lower bound on a collateral asset's initial weight (when the collateral is first whitelisted) is different from that enforced if the weight is updated; this discrepancy increases the likelihood of collateral seizures by liquidators.

A collateral asset's weight represents the level of risk associated with accepting that asset as collateral. This risk calculation comes into play when the protocol is assessing whether a liquidator can seize a user's non-UA collateral. To determine the value of each collateral asset, the protocol multiplies the user's balance of that asset by the collateral weight (a percentage). A riskier asset will have a lower weight and thus a lower value. If the total value of a user's non-UA collateral is less than the user's UA debt, a liquidator can seize the collateral.

When whitelisting a collateral asset, the `Perpetual.addWhiteListedCollateral` function requires the collateral weight to be between 10% and 100% (figure 2.1). According to the documentation, these are the correct bounds for a collateral asset's weight.

```
function addWhiteListedCollateral(
    IERC20Metadata asset,
    uint256 weight,
    uint256 maxAmount
) public override onlyRole(GOVERNANCE) {
    if (weight < 1e17) revert Vault_InsufficientCollateralWeight();
    if (weight > 1e18) revert Vault_ExcessiveCollateralWeight();
    [...]
}
```

Figure 2.1: A snippet of the `addWhiteListedCollateral` function in `Vault.sol#L224-230`

However, governance can choose to update that weight via a call to `Perpetual.changeCollateralWeight`, which allows the weight to be between 1% and 100% (figure 2.2).

```
function changeCollateralWeight(IERC20Metadata asset, uint256 newWeight) external
override onlyRole(GOVERNANCE) {
    uint256 tokenIdx = tokenToCollateralIdx[asset];
    if (!((tokenIdx != 0) || (address(asset) == address(UA)))) revert
Vault_UnsupportedCollateral();

    if (newWeight < 1e16) revert Vault_InsufficientCollateralWeight();
    if (newWeight > 1e18) revert Vault_ExcessiveCollateralWeight();
    [...]
}
```

Figure 2.2: A snippet of the `changeCollateralWeight` function in `Vault.sol#L254-259`

If the weight of a collateral asset were mistakenly set to less than 10%, the value of that collateral would decrease, thereby increasing the likelihood of seizures of non-UA collateral.

Fix Analysis

This issue has been resolved. The protocol now enforces the same lower bound when setting and updating a collateral asset's weight.

3. Solidity compiler optimizations can be problematic

Status: **Unresolved**	
Severity: **Informational**	Difficulty: **High**
Type: Undefined Behavior	Finding ID: TOB-INC-3
Target: Increment Protocol	

Description
The Increment Protocol contracts have enabled optional compiler optimizations in Solidity.

There have been several optimization bugs with security implications. Moreover, optimizations are actively being developed. Solidity compiler optimizations are disabled by default, and it is unclear how many contracts in the wild actually use them. Therefore, it is unclear how well they are being tested and exercised.

Security issues due to optimization bugs have occurred in the past. A medium- to high-severity bug in the Yul optimizer was introduced in Solidity version 0.8.13 and was fixed only recently, in Solidity version 0.8.17. Another medium-severity optimization bug—one that caused memory writes in inline assembly blocks to be removed under certain conditions—was patched in Solidity 0.8.15.

A compiler audit of Solidity from November 2018 concluded that the optional optimizations may not be safe.

It is likely that there are latent bugs related to optimization and that new bugs will be introduced due to future optimizations.

Fix Analysis
This issue has not been resolved. The Increment Finance team is willing to accept the risk of optimization-related bugs. The team provided the following additional context: “We are aware of the risk associated with using an untested compiler and plan to limit the risk using the principles of a protected launch.”

4. Support for multiple reserve tokens allows for arbitrage	
Status: **Unresolved**	
Severity: **Informational**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-INC-4
Target: `contracts/tokens/UA.sol`	

Description

Because the UA token contract supports multiple reserve tokens, it can be used to swap one reserve token for another at a ratio of 1:1. This creates an arbitrage opportunity, as it enables users to swap reserve tokens with different prices.

Users can deposit supported reserve tokens in the UA contract in exchange for UA tokens at a 1:1 ratio (figure 4.1).

```
function mintWithReserve(uint256 tokenIdx, uint256 amount) external override {
    // Check that the reserve token is supported
    if (tokenIdx > reserveTokens.length - 1) revert UA_InvalidReserveTokenIndex();
    ReserveToken memory reserveToken = reserveTokens[tokenIdx];

    // Check that the cap of the reserve token isn't reached
    uint256 wadAmount = LibReserve.tokenToWad(reserveToken.asset.decimals(),
amount);
    if (reserveToken.currentReserves + wadAmount > reserveToken.mintCap) revert
UA_ExcessiveTokenMintCapReached();

    _mint(msg.sender, wadAmount);
    reserveTokens[tokenIdx].currentReserves += wadAmount;

    reserveToken.asset.safeTransferFrom(msg.sender, address(this), amount);
}
```

Figure 4.1: The `mintWithReserve` function in `UA.sol#L38-51`

Similarly, users can withdraw the amount of a deposit by returning their UA in exchange for any supported reserve token, also at a 1:1 ratio (figure 4.2).

```
function withdraw(uint256 tokenIdx, uint256 amount) external override {
    // Check that the reserve token is supported
    if (tokenIdx > reserveTokens.length - 1) revert UA_InvalidReserveTokenIndex();
    IERC20Metadata reserveTokenAsset = reserveTokens[tokenIdx].asset;
    _burn(msg.sender, amount);
    reserveTokens[tokenIdx].currentReserves -= amount;

    uint256 tokenAmount = LibReserve.wadToToken(reserveTokenAsset.decimals(),
amount);
    reserveTokenAsset.safeTransfer(msg.sender, tokenAmount);
}
```

Figure 4.2: The `withdraw` function in `UA.sol#L56-66`

Thus, a user could mint UA by depositing a less valuable reserve token and then withdraw the same amount of a more valuable token in one transaction, engaging in arbitrage.

Fix Analysis

This issue has not been resolved. The Increment Finance team is aware of the arbitrage opportunity and does not plan on mitigating it, as mitigation could increase the code's complexity.

5. Ownership transfers can be front-run	
Status: **Resolved**	
Severity: **High**	Difficulty: **High**
Type: Timing	Finding ID: TOB-INC-5
Target: `contracts/utils/PerpOwnable.sol`	

Description

The `PerpOwnable` contract provides an access control mechanism for the minting and burning of a `Perpetual` contract's vBase or vQuote tokens. The owner of these token contracts is set via the `transferPerpOwner` function, which assigns the owner's address to the `perp` state variable. This function is designed to be called only once, during deployment, to set the `Perpetual` contract as the owner of the tokens. Then, as the tokens' owner, the `Perpetual` contract can mint / burn tokens during liquidity provisions, trades, and liquidations. However, because the function is external, anyone can call it to set his or her own malicious address as `perp`, taking ownership of a contract's vBase or vQuote tokens.

```
function transferPerpOwner(address recipient) external {
    if (recipient == address(0)) revert PerpOwnable_TransferZeroAddress();
    if (perp != address(0)) revert PerpOwnable_OwnershipAlreadyClaimed();

    perp = recipient;
    emit PerpOwnerTransferred(msg.sender, recipient);
}
```

Figure 5.1: The `transferPerpOwner` function in `PerpOwnable.sol#L29-L35`

If the call were front-run, the `Perpetual` contract would not own the vBase or vQuote tokens, and any attempts to mint / burn tokens would revert. Since all user interactions require the minting or burning of tokens, no liquidity provisions, trades, or liquidations would be possible; the market would be effectively unusable. An attacker could launch such an attack upon every perpetual market deployment to cause a denial of service (DoS).

Fix Analysis

This issue has been resolved. The `transferPerpOwner` function can now be called by only the deployer of a given `Perpetual` contract.

6. Funding payments are made in the wrong token

Status: **Resolved**	
Severity: **High**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-INC-6
Target: `contracts/ClearingHouse.sol`	

Description

The funding payments owed to users are made in vBase instead of UA tokens; this results in incorrect calculations of users' profit-and-loss (PnL) values, an increased risk of liquidations, and a delay in the convergence of a `Perpetual` contract's value with that of the underlying base asset.

When the protocol executes a trade or liquidity provision, one of its first steps is settling the funding payments that are due to the calling user. To do that, it calls the `_settleUserFundingPayments` function in the `ClearingHouse` contract (figure 6.1). The function sums the funding payments due to the user (as a trader and / or a liquidity provider) across all perpetual markets. Once the function has determined the final funding payment due to the user (`fundingPayments`), the `Vault` contract's `settlePnL` function changes the UA balance of the user.

```
function _settleUserFundingPayments(address account) internal {
    int256 fundingPayments;
    uint256 numMarkets = getNumMarkets();
    for (uint256 i = 0; i < numMarkets; ) {
        fundingPayments += perpetuals[i].settleTrader(account) +
perpetuals[i].settleLp(account);

        unchecked {
            ++i;
        }
    }

    if (fundingPayments != 0) {
        vault.settlePnL(account, fundingPayments);
    }
}
```

Figure 6.1: The `_settleUserFundingPayments` function in `ClearingHouse.sol#L637-651`

Both the `Perpetual.settleTrader` and `Perpetual.settleLp` functions internally call `_getFundingPayments` to calculate the funding payment due to the user for a given market (figure 6.2).

```
function _getFundingPayments(
    bool isLong,
    int256 userCumFundingRate,
    int256 globalCumFundingRate,
    int256 vBaseAmountToSettle
) internal pure returns (int256 upcomingFundingPayment) {
    [...]
    if (userCumFundingRate != globalCumFundingRate) {
        int256 upcomingFundingRate = isLong
            ? userCumFundingRate - globalCumFundingRate
            : globalCumFundingRate - userCumFundingRate;

        // fundingPayments = fundingRate * vBaseAmountToSettle
        upcomingFundingPayment = upcomingFundingRate.wadMul(vBaseAmountToSettle);
    }
}
```

Figure 6.2: The `_getFundingPayments` function in `Perpetual.sol#L1152-1173`

However, the `upcomingFundingPayment` value is expressed in vBase, since it is the product of a percentage, which is unitless, and a vBase token amount, `vBaseAmountToSettle`. Thus, the `fundingPayments` value that is calculated in `_settleUserFundingPayments` is also expressed in vBase. However, the `settlePnL` function internally updates the user's balance of UA, not vBase. As a result, the user's UA balance will be incorrect, since the user's profit or loss may be significantly higher or lower than it should be. This discrepancy is a function of the price difference between the vBase and UA tokens.

The use of vBase tokens for funding payments causes three issues. First, when withdrawing UA tokens, the user may lose or gain much more than expected. Second, since the UA balance affects the user's collateral reserve total, the balance update may increase or decrease the user's risk of liquidation. Finally, since funding payments are not made in the notional asset, the convergence between the mark and index prices may be delayed.

Fix Analysis

This issue has been resolved. The protocol now uses the `vBase.indexPrice()` function to convert vBase token amounts to UA and thus makes funding payments in UA instead of vBase.

7. Excessive dust collection may lead to premature closures of long positions

Status: **Resolved**	
Severity: **Medium**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-INC-7
Target: `contracts/Perpetual.sol`	

Description

The upper bound on the amount of funds considered dust by the protocol may lead to the premature closure of long positions.

The protocol collects dust to encourage complete closures instead of closures that leave a position with a small balance of vBase. One place that dust collection occurs is the `Perpetual` contract's `_reducePositionOnMarket` function (figure 7.1).

```
function _reducePositionOnMarket(
    LibPerpetual.TraderPosition memory user,
    bool isLong,
    uint256 proposedAmount,
    uint256 minAmount
)
    internal
    returns (
        int256 baseProceeds,
        int256 quoteProceeds,
        int256 addedOpenNotional,
        int256 pnl
    )
{
    int256 positionSize = int256(user.positionSize);

    uint256 bought;
    uint256 feePer;
    if (isLong) {
        quoteProceeds = -(proposedAmount.toInt256());
        (bought, feePer) = _quoteForBase(proposedAmount, minAmount);
        baseProceeds = bought.toInt256();
    } else {
        (bought, feePer) = _baseForQuote(proposedAmount, minAmount);
        quoteProceeds = bought.toInt256();
        baseProceeds = -(proposedAmount.toInt256());
    }
    int256 netPositionSize = baseProceeds + positionSize;
    if (netPositionSize > 0 && netPositionSize <= 1e17) {
        _donate(netPositionSize.toUint256());
        baseProceeds -= netPositionSize;
    }
    [...]
}
```

Figure 7.1: The `_reducePositionOnMarket` function in `Perpetual.sol#L876-921`

If `netPositionSize`, which represents a user's position after its reduction, is between 0 and 1e17 (1/10 of an 18-decimal token), the system will treat the position as closed and donate the dust to the insurance protocol. This will occur regardless of whether the user intended to reduce, rather than fully close, the position. (Note that `netPositionSize` is positive if the overall position is long. The dust collection mechanism used for short positions is discussed in TOB-INC-11.)

However, if `netPositionSize` is tracking a high-value token, the donation to `Insurance` will no longer be insignificant; 1/10 of 1 vBTC, for instance, would be worth ~USD 2,000 (at the time of writing). Thus, the donation of a user's vBTC dust (and the resultant closure of the vBTC position) could prevent the user from profiting off of a ~USD 2,000 position.

Fix Analysis

This issue has been resolved. The Increment Finance team updated the dust collection mechanism, limiting the amount of dust collected from a position to USD 0.10.

8. Problematic use of primitive operations on fixed-point integers

Status: **Resolved**	
Severity: **Informational**	Difficulty: **High**
Type: Undefined Behavior	Finding ID: TOB-INC-8
Target: `lib/LibMath.sol`	

Description

The protocol's use of primitive operations over fixed-point signed and unsigned integers increases the risk of overflows and undefined behavior.

The Increment Protocol uses the `PRBMathSD59x18` and `PRBMathUD60x18` math libraries to perform operations over 59x18 signed integers and 60x18 unsigned integers, respectively (specifically to perform multiplication and division and to find their absolute values). These libraries aid in calculations that involve percentages or ratios or require decimal precision.

When a smart contract system relies on primitive integers and fixed-point ones, it should avoid arithmetic operations that involve the use of both types. For example, using `x.wadMul(y)` to multiply two fixed-point integers will provide a different result than using `x * y`. For that reason, great care must be taken to differentiate between variables that are fixed-point and those that are not. Calculations involving fixed-point values should use the provided library operations; calculations involving both fixed-point and primitive integers should be avoided unless one type is converted to the other.

However, a number of multiplication and division operations in the codebase use both primitive and fixed-point integers. These include those used to calculate the new time-weighted average prices (TWAPs) of index and market prices (figure 8.1).

```
function _updateTwap() internal {
    uint256 currentTime = block.timestamp;
    int256 timeElapsed = (currentTime - globalPosition.timeOfLastTrade).toInt256();

    /*
        priceCumulative1 = priceCumulative0 + price1 * timeElapsed
    */

    // will overflow in ~3000 years
    // update cumulative chainlink price feed
    int256 latestChainlinkPrice = indexPrice();
    oracleCumulativeAmount += latestChainlinkPrice * timeElapsed;
    // update cumulative market price feed
    int256 latestMarketPrice = marketPrice().toInt256();
    marketCumulativeAmount += latestMarketPrice * timeElapsed;

    uint256 timeElapsedSinceBeginningOfPeriod = block.timestamp -
globalPosition.timeOfLastTwapUpdate;

    if (timeElapsedSinceBeginningOfPeriod >= twapFrequency) {
        /*
            TWAP = (priceCumulative1 - priceCumulative0) / timeElapsed
        */

        // calculate chainlink twap
        oracleTwap = ((oracleCumulativeAmount -
oracleCumulativeAmountAtBeginningOfPeriod) /
            timeElapsedSinceBeginningOfPeriod.toInt256()).toInt128();

        // calculate market twap
        marketTwap = ((marketCumulativeAmount -
marketCumulativeAmountAtBeginningOfPeriod) /
            timeElapsedSinceBeginningOfPeriod.toInt256()).toInt128();

        // reset cumulative amount and timestamp
        oracleCumulativeAmountAtBeginningOfPeriod = oracleCumulativeAmount;
        marketCumulativeAmountAtBeginningOfPeriod = marketCumulativeAmount;
        globalPosition.timeOfLastTwapUpdate = block.timestamp.toUint64();

        emit TwapUpdated(oracleTwap, marketTwap);
    }
}
```

Figure 8.1: The `_updateTwap` function in `Perpetual.sol#L1071-1110`

Similarly, the `_getUnrealizedPnL` function in the `Perpetual` contract calculates the `tradingFees` value by multiplying a primitive and a fixed-point integer (figure 8.2).

```
function _getUnrealizedPnL(LibPerpetual.TraderPosition memory trader) internal view
returns (int256) {
    int256 oraclePrice = indexPrice();
    int256 vQuoteVirtualProceeds = int256(trader.positionSize).wadMul(oraclePrice);
    int256 tradingFees = (vQuoteVirtualProceeds.abs() * market.out_fee().toInt256())
/ CURVE_TRADING_FEE_PRECISION; // @dev: take upper bound on the trading fees

    // in the case of a LONG, trader.openNotional is negative but
vQuoteVirtualProceeds is positive
    // in the case of a SHORT, trader.openNotional is positive while
vQuoteVirtualProceeds is negative
    return int256(trader.openNotional) + vQuoteVirtualProceeds - tradingFees;
}
```

Figure 8.2: The `_getUnrealizedPnL` function in `Perpetual.sol#L1175-1183`

These calculations can lead to unexpected overflows or cause the system to enter an undefined state. Note that there are other such calculations in the codebase that are not documented in this finding.

Fix Analysis

This issue has been resolved. The Increment Finance team updated the inline documentation to explain the use of primitive division and multiplication involving both fixed-point and non-fixed-point integers.

9. Liquidations are vulnerable to sandwich attacks	
Status: **Resolved**	
Severity: **Medium**	Difficulty: **High**
Type: Timing	Finding ID: TOB-INC-9
Target: `contracts/ClearingHouse.sol`	

Description

Token swaps that are performed to liquidate a position use a hard-coded zero as the "minimum-amount-out" value, making them vulnerable to sandwich attacks.

The minimum-amount-out value indicates the minimum amount of tokens that a user will receive from a swap. The value is meant to provide protection against pool illiquidity and sandwich attacks. Senders of position and liquidity provision updates are allowed to specify a minimum amount out. However, the minimum-amount-out value used in liquidations of both traders' and liquidity providers' positions is hard-coded to zero. Figures 9.1 and 9.2 show the functions that perform these liquidations (`_liquidateTrader` and `_liquidateLp`, respectively).

```
function _liquidateTrader(
    uint256 idx,
    address liquidatee,
    uint256 proposedAmount
) internal returns (int256 pnL, int256 positiveOpenNotional) {
    (positiveOpenNotional) = int256(_getTraderPosition(idx,
liquidatee).openNotional).abs();

    LibPerpetual.Side closeDirection = _getTraderPosition(idx,
liquidatee).positionSize >= 0
        ? LibPerpetual.Side.Short
        : LibPerpetual.Side.Long;

    // (liquidatee, proposedAmount)
    (, , pnL, ) = perpetuals[idx].changePosition(liquidatee, proposedAmount, 0,
closeDirection, true);

    // traders are allowed to reduce their positions partially, but liquidators have
to close positions in full
    if (perpetuals[idx].isTraderPositionOpen(liquidatee))
        revert ClearingHouse_LiquidateInsufficientProposedAmount();

    return (pnL, positiveOpenNotional);
}
```

Figure 9.1: The `_liquidateTrader` function in `ClearingHouse.sol#L522-541`

```
function _liquidateLp(
    uint256 idx,
    address liquidatee,
    uint256 proposedAmount
) internal returns (int256 pnL, int256 positiveOpenNotional) {
    positiveOpenNotional = _getLpOpenNotional(idx, liquidatee).abs();

    // close lp
    (pnL, , ) = perpetuals[idx].removeLiquidity(
        liquidatee,
        _getLpLiquidity(idx, liquidatee),
        [uint256(0), uint256(0)],
        proposedAmount,
        0,
        true
    );
    _distributeLpRewards(idx, liquidatee);

    return (pnL, positiveOpenNotional);
}
```

Figure 9.2: The `_liquidateLp` function in `ClearingHouse.sol#L543-562`

Without the ability to set a minimum amount out, liquidators are not guaranteed to receive any tokens from the pool during a swap. If a liquidator does not receive the correct amount of tokens, he or she will be unable to close the position, and the transaction will revert; the revert will also prolong the Increment Protocol's exposure to debt. Moreover, liquidators will be discouraged from participating in liquidations if they know that they may be subject to sandwich attacks and may lose money in the process.

Fix Analysis

This issue has been resolved. Liquidators can now specify a minimum-amount-out value when liquidating a position.

10. Accuracy of market and oracle TWAPs is tied to the frequency of user interactions

Status: **Resolved**	
Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-INC-10
Target: `contracts/ClearingHouse.sol`	

Description

The oracle and market TWAPs can be updated only during traders' and liquidity providers' interactions with the protocol; a downtick in user interactions will result in less accurate TWAPs that are more susceptible to manipulation.

The accuracy of a TWAP is related to the number of data points available for the average price calculation. The less often prices are logged, the less robust the TWAP becomes. In the case of the Increment Protocol, a TWAP can be updated with each block that contains a trader or liquidity provider interaction. However, during a market slump (i.e., a time of reduced network traffic), there will be fewer user interactions and thus fewer price updates.

TWAP updates are performed by the `Perpetual._updateTwap` function, which is called by the internal `Perpetual._updateGlobalState` function. Other protocols, though, take a different approach to keeping markets up to date. The Compound Protocol, for example, has an `accrueInterest` function that is called upon every user interaction but is *also* a standalone public function that anyone can call.

Fix Analysis

This issue has been resolved. The Increment Finance team created a public `updateGlobalState` function that anyone can call to update a perpetual market.

11. Liquidations of short positions may fail because of insufficient dust collection

Status: **Resolved**	
Severity: **Low**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-INC-11
Target: `contracts/Perpetual.sol`	

Description

Because the protocol does not collect the dust associated with short positions, attempts to fully close and then liquidate those positions will fail.

One of the key requirements for the successful liquidation of a position is the closure of the entire position; in other words, by the end of the transaction, the debt and assets of the trader or liquidity provider must be zero. The process of closing a long position is a straightforward one, since identifying the correct `proposedAmount` value (the amount of tokens to be swapped) is trivial. Finding the correct `proposedAmount` for a short position, however, is more complex.

If the `proposedAmount` estimate is incorrect, the transaction will result in leftover dust, which the protocol will attempt to collect (figure 11.1).

```
function _reducePositionOnMarket(
    LibPerpetual.TraderPosition memory user,
    bool isLong,
    uint256 proposedAmount,
    uint256 minAmount
)
    internal
    returns (
        int256 baseProceeds,
        int256 quoteProceeds,
        int256 addedOpenNotional,
        int256 pnl
    )
{
    int256 positionSize = int256(user.positionSize);

    uint256 bought;
    uint256 feePer;
    if (isLong) {
        quoteProceeds = -(proposedAmount.toInt256());
        (bought, feePer) = _quoteForBase(proposedAmount, minAmount);
        baseProceeds = bought.toInt256();
    } else {
        (bought, feePer) = _baseForQuote(proposedAmount, minAmount);
        quoteProceeds = bought.toInt256();
        baseProceeds = -(proposedAmount.toInt256());
    }

    int256 netPositionSize = baseProceeds + positionSize;
    if (netPositionSize > 0 && netPositionSize <= 1e17) {
        _donate(netPositionSize.toUint256());
        baseProceeds -= netPositionSize;
    }
    [...]
}
```

Figure 11.1: The `_reducePositionOnMarket` function in `Perpetual.sol#L876-921`

The protocol will collect leftover dust only if `netPositionSize` is greater than zero, which is possible only if the position that is being closed is a long one. If a short position is left with any dust, it will not be collected, since `netPositionSize` will be less than zero.

This inconsistency has a direct impact on the success of liquidations, because a position must be completely closed in order for a liquidation to occur; no dust can be left over. When liquidating the position of a liquidity provider, the `Perpetual` contract's `_settleLpPosition` function checks whether `netBasePosition` is less than zero (as shown in figure 11.2). If it is, the liquidation will fail. Because the protocol does not collect dust from short positions, the `netBasePosition` value of such a position may be less than zero. The `ClearingHouse._liquidateTrader` function, called to liquidate traders' positions, enforces a similar requirement regarding total closures.

```
function _settleLpPosition(
    LibPerpetual.TraderPosition memory positionToClose,
    uint256 proposedAmount,
    uint256 minAmount,
    bool isLiquidation
) internal returns (int256 pnl, int256 quoteProceeds) {
    int256 baseProceeds;

    (baseProceeds, quoteProceeds, , pnl) = _reducePositionOnMarket(
        positionToClose,
        !(positionToClose.positionSize > 0),
        proposedAmount,
        minAmount
    );
    [...]
    int256 netBasePosition = positionToClose.positionSize + baseProceeds;

    if (netBasePosition < 0) revert Perpetual_LPOpenPosition();
    if (netBasePosition > 0 && netBasePosition <= 1e17)
_donate(netBasePosition.toUint256());
}
```

Figure 11.2: The `_settleLpPosition` function in `Perpetual.sol#L1005-1030`

If the liquidation of a position fails, any additional attempts at liquidation will lower the liquidator's profit margin, which might dissuade the liquidator from trying again. Additionally, failed liquidations prolong the protocol's exposure to debt.

Exploit Scenario

Alice, a liquidator, notices that a short position is no longer valid and decides to liquidate it. However, Alice sets an incorrect `proposedAmount` value, so the position is left with some dust. Because the protocol does not collect the dust of short positions, the liquidation fails. As a result, Alice loses money—and the loss dissuades her from attempting to liquidate any other undercollateralized positions.

Fix Analysis

This issue has been resolved. The Increment Finance team updated the liquidation flows for both short and long positions to ensure that liquidations of short positions will also succeed.

12. Project dependencies contain vulnerabilities	
Status: **Resolved**	
Severity: **Low**	Difficulty: **High**
Type: Patching	Finding ID: TOB-INC-12
Target: `increment-protocol`	

Description
Although dependency scans did not identify a direct threat to the project under review, `yarn audit` identified dependencies with known vulnerabilities. Due to the sensitivity of the deployment code and its environment, it is important to ensure that dependencies are not malicious. Problems with dependencies in the JavaScript community could have a significant effect on the repository under review. The output below details the high-severity vulnerabilities:

CVE ID	Description	Dependency
CVE-2021-23358	Arbitrary code injection vulnerability	`underscore`
CVE-2021-43138	Prototype pollution	`async`
CVE-2021-23337	Command injection vulnerability	`lodash`
CVE-2022-0235	"`node-fetch` is vulnerable to exposure of sensitive information to an unauthorized actor"	`node-fetch`

Figure 12.1: Advisories affecting `increment-protocol` dependencies

Fix Analysis
This issue has been resolved. The Increment Finance team updated most of the dependencies to their latest versions and implemented an automated dependency-monitoring solution.

13. Risks associated with oracle outages	
Status: **Unresolved**	
Severity: **Informational**	Difficulty: **High**
Type: Configuration	Finding ID: TOB-INC-13
Target: `increment-protocol`	

Description

Under extreme market conditions, the Chainlink oracle may cease to work as expected, causing unexpected behavior in the Increment Protocol.

Such oracle issues have occurred in the past. For example, during the LUNA market crash, the Venus protocol was exploited because Chainlink stopped providing up-to-date prices. The interruption occurred because the price of LUNA dropped below the minimum price (`minAnswer`) allowed by the LUNA / USD price feed on the BNB chain. As a result, all oracle updates reverted. Chainlink's automatic circuit breakers, which pause price feeds during extreme market conditions, could pose similar problems.

Note that these kinds of events cannot be tracked on-chain. If a price feed is paused, `updatedAt` will still be greater than zero, and `answeredInRound` will still be equal to `roundID`.

Thus, the Increment Finance team should implement an off-chain monitoring solution to detect any anomalous behavior exhibited by Chainlink oracles. The monitoring solution should check for the following conditions and issue alerts if they occur, as they may be indicative of abnormal market events:

- An asset price that is approaching the `minAnswer` or `maxAnswer` value
- The suspension of a price feed by an automatic circuit breaker
- Any large deviations in the price of an asset

Fix Analysis

This issue remains unresolved. However, the Increment Finance team indicated that it will be implementing an off-chain monitoring solution in the future.

A. Status Categories

The following table describes the statuses used to indicate whether an issue has been sufficiently addressed.

Fix Status

Status	Description
Undetermined	The status of the issue was not determined during this engagement.
Unresolved	The issue persists and has not been resolved.
Partially Resolved	The issue persists but has been partially resolved.
Resolved	The issue has been sufficiently resolved.

B. Vulnerability Categories

The following tables describe the vulnerability categories, severity levels, and difficulty levels used in this document.

Vulnerability Categories	
Category	Description
Access Controls	Insufficient authorization or assessment of rights
Auditing and Logging	Insufficient auditing of actions or logging of problems
Authentication	Improper identification of users
Configuration	Misconfigured servers, devices, or software components
Cryptography	A breach of system confidentiality or integrity
Data Exposure	Exposure of sensitive information
Data Validation	Improper reliance on the structure or values of data
Denial of Service	A system failure with an availability impact
Error Reporting	Insecure or insufficient reporting of error conditions
Patching	Use of an outdated software package or library
Session Management	Improper identification of authenticated users
Testing	Insufficient test methodology or test coverage
Timing	Race conditions or other order-of-operations flaws
Undefined Behavior	Undefined behavior triggered within the system

Severity Levels	
Severity	**Description**
Informational	The issue does not pose an immediate risk but is relevant to security best practices.
Undetermined	The extent of the risk was not determined during this engagement.
Low	The risk is small or is not one the client has indicated is important.
Medium	User information is at risk; exploitation could pose reputational, legal, or moderate financial risks.
High	The flaw could affect numerous users and have serious reputational, legal, or financial implications.

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploitation was not determined during this engagement.
Low	The flaw is well known; public tools for its exploitation exist or can be scripted.
Medium	An attacker must write an exploit or will need in-depth knowledge of the system.
High	An attacker must have privileged access to the system, may need to know complex technical details, or must discover other weaknesses to exploit this issue.