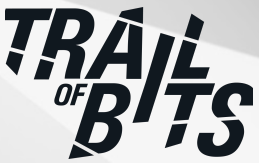

Increment Finance: Increment Protocol

Security Assessment

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 80+ 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.

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 timeline, test targets, and coverage are provided in subsequent sections of this report.

Project Scope

Our testing efforts were focused on the identification of flaws that could result in a compromise of confidentiality, integrity, or availability of the target system. We conducted this audit with full knowledge of the target system, including access to the source code and relevant documentation. We performed static testing of the target system and its codebase, using both automated and manual processes.

Summary of Findings

The audit uncovered significant flaws that could impact system confidentiality, integrity, or availability. A summary of the findings and details on notable findings are 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

Notable Findings

Significant flaws that impact system confidentiality, integrity, or availability are listed below.

- **TOB-INC-1**
  The governance role constitutes a single point of failure because of the large number of privileges granted to it.

- **TOB-INC-5**
  Because the `transferPerpOwner` function can be called by anyone, an attacker could front-run a legitimate call to the function to cause a denial of service (DoS).

- **TOB-INC-6**
  Funding payments are made in vBase tokens; however, when a user is owed a funding payment, the protocol updates the user's balance of UA tokens. This results in the incorrect calculation of the user's profits and losses, a delayed convergence between the value of a `Perpetual` contract and that of the underlying asset, and an increased risk of liquidations.

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 Goals

The engagement was scoped to provide a security assessment of Increment Finance's Increment Protocol. Specifically, we sought to answer the following non-exhaustive list of questions:

- Could an attacker steal funds from the system?
- Are there appropriate access controls in place?
- Are the free-collateral and margin requirements calculated correctly?
- Are user-provided parameters sufficiently validated?
- Are the arithmetic calculations and state changes performed during position updates, liquidity provisions, and liquidations correct?
- Does the dust collection mechanism lead to any undefined behavior?
- Are there front-running or DoS opportunities in the system?
- How is oracle data obtained and handled?
- Is it possible to prevent the execution of liquidations?

Project Targets

The engagement involved a review and testing of the following target.

Increment Protocol

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

Project Coverage

This section provides an overview of the analysis coverage of the review, as determined by our high-level engagement goals. Our approaches and their results include the following:

`ClearingHouse` and `Perpetual`. The `ClearingHouse` contract is the entry point for all user interactions with the protocol. It interacts with the various perpetual markets and the vault where funds are stored. The `Perpetual` contract represents a single perpetual market and contains the logic for deposits, withdrawals, liquidity provisions, liquidations, and interactions with the Curve virtual automated market maker (vAMM).

- In our manual review of the contracts, we reviewed the correctness of the following arithmetic calculations and state changes:
    - Those that occur when a trader increases, reduces, reverses, or closes his or her position
    - Those that occur when a liquidity provider supplies or removes liquidity
    - Those that occur when a liquidator attempts to liquidate a trader's or liquidity provider's position
- We also analyzed the calculation of the funding payments owed to a user. This led us to discover that those payments are made in vBase tokens instead of UA tokens (TOB-INC-6).
- We reviewed the arithmetic calculations that determine a user's free collateral to ensure that they comply with the mathematical specification provided by the client.
- We checked the protocol for ways to front-run a contract's initialization, a liquidation or liquidity provision operation, or a position update. This led us to find that liquidators' swaps are vulnerable to sandwich attacks because the minimum payout amount is hard-coded to zero (TOB-INC-9).
- We reviewed the protocol's dust collection mechanism and found that a position may be closed prematurely if an excessive amount of dust is collected (TOB-INC-7). Additionally, as detailed in TOB-INC-11, because dust is not collected when short positions are closed, attempts to liquidate those positions may fail.

`Vault`. The `Vault` contract holds user funds across all perpetual markets. We conducted a manual review of this contract and investigated the following:

- We checked whether an attacker could steal funds from the `Vault` contract by reentering any functions in the `ClearingHouse` or `Perpetual` contract.
- We checked whether an attacker could bypass any of the access controls on `Vault` functions to steal funds from the contract.
- We reviewed the arithmetic operations and state changes that occur within the `Vault` to ensure that user balances are monitored correctly and that token conversions and transfers are performed correctly.
- We reviewed the use of ERC4626 tokens as collateral and checked whether it introduces any undefined behavior.
- We reviewed the bounds on all owner-controlled system parameters and found that the bounds on the collateral weight parameter are inconsistent (TOB-INC-2).

`Oracle`. The `Oracle` contract is responsible for reporting underlying currency prices provided by Chainlink oracles. We conducted a manual review of this contract and investigated the following:

- We reviewed whether the contract adheres to best practices regarding the retrieval and validation of Chainlink pricing data.
- We reviewed the mechanism for retrieving ERC4626 token prices to ensure that it adheres to the token standard and is invulnerable to manipulation.
- We reviewed the way in which the protocol tracks sequencer uptime to ensure that if the sequencer goes down, the contract will not report stale or incorrect pricing data.

`Insurance`. If a user's or liquidity provider's position enters default, the protocol can draw funds from the `Insurance` contract to avoid going into debt. We conducted a manual review of this contract and investigated the following:

- We checked whether an attacker could bypass any of the access controls on `Insurance` functions to steal funds from the protocol.
- We reviewed the arithmetic operations and state changes performed when insurance must kick in to prevent the protocol from going into debt or incurring an unexpected loss.

`CurveCryptoViews`. The `CurveCryptoViews` contract is a view-only contract used to estimate the amount of fees associated with a swap or the amount of output or input

tokens consumed during a swap. We manually reviewed the protocol's Solidity implementation of Curve's `get_dy` function, `get_dy_ex_fees`, to check whether it preserves the original functionality.

`tokens/`. The `tokens/` folder holds the `VBase`, `VQuote`, and `UA` contracts. A pair of vBase and vQuote tokens is used to represent each perpetual market. The `UA` contract serves as the unit-of-account token for a user's profits and losses. We conducted a manual review of these contracts and investigated the following:

- We reviewed the issuance and redemption mechanism of the `UA` contract. This led to the discovery of the arbitrage risk detailed in TOB-INC-4. Specifically, an arbitrageur could leverage the price difference between two tokens to make a risk-free profit.
- We reviewed the `VBase` contract's price retrieval mechanism to ensure that it adheres to best practices regarding the retrieval and validation of Chainlink pricing data.
- We reviewed the `BaseToken` contract, checking whether the omissions from the original Solmate implementation lead to any undefined behavior.

`lib/`. The `lib/` folder holds the `LibMath`, `LibPerpetual`, and `LibReserve` libraries. The `LibMath` library contains arithmetic functions that use `PRBMath` under the hood for fixed-point arithmetic. The `LibReserve` library contains the logic for converting a token's decimal precision to (or from) 18-decimal precision. The `LibPerpetual` library contains various structs used across the codebase (and no state-changing logic). We conducted a manual review of these libraries and investigated the following:

- We reviewed the codebase's use of fixed-point integers and operations. This review yielded one finding, TOB-INC-8, which concerns the use of primitive operations on fixed-point integers as well as arithmetic calculations that combine primitive and fixed-point integers.
- We manually verified that the `tokenToWad` and `wadToToken` functions are symmetric operations.

`utils/`. The `utils/` folder holds the `IncreAccessControl` and `PerpOwnable` contracts. The `IncreAccessControl` contract defines modifiers for checking whether a sender has the governance or manager role. The `PerpOwnable` contract is inherited by the `Perpetual` contract and represents the owner of a given perpetual market. We conducted a manual review of these contracts and investigated the following:

- We reviewed the contracts' access controls. This led to the discovery of TOB-INC-1, which highlights the excessive privileges given to the governance role.
- We looked for ways in which a user could take ownership of a perpetual market. This led us to discover that a malicious user could front-run a call to `transferPerpOwner` to cause a DoS (TOB-INC-5).

zkSync. Because the protocol will be deployed on zkSync, we looked for any edge cases that could result from the deployment of Solidity smart contracts on zkSync.

Coverage Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. The following list outlines the coverage limitations of the engagement and indicates system elements that may warrant further review:

- **`PRBMathSD59x18` and `PRBMathUD60x18`.** The protocol uses the two `PRBMath` libraries to perform arithmetic operations over fixed-point signed and unsigned integers. We did not review this external library during the audit.
- **`ClearingHouseViewer`.** The protocol's front end uses `ClearingHouseViewer`, a view-only contract, to gauge the state of the system. This contract was excluded from the audit's scope at Increment Finance's request.
- **`CurveCryptoViews`.** Only the `get_dy_ex_fees` function was reviewed during the audit.
- **`LibReserve`.** Because this library holds only structs (and not state-changing logic or critical view functions), we considered it out of scope.
- **Dynamic testing.** Because of the time constraints of the audit, we were unable to perform end-to-end dynamic fuzz testing of the system.

Codebase Maturity Evaluation

Trail of Bits uses a traffic-light protocol to provide each client with a clear understanding of the areas in which its codebase is mature, immature, or underdeveloped. Deficiencies identified here often stem from root causes within the software development life cycle that should be addressed through standardization measures (e.g., the use of common libraries, functions, or frameworks) or training and awareness programs.

Category	Summary	Result
Arithmetic	The codebase uses Solidity v0.8.15 for arithmetic operations and the `PRBMathSD59x18` and `PRBMathUD60x18` libraries for fixed-point arithmetic. We were provided a mathematical specification discussing critical operations. However, we found issues regarding the collection of dust (TOB-INC-7), which can be excessive, and the incorrect use of fixed-point and normal integers in the same primitive operations (TOB-INC-8). We recommend using Echidna for dynamic fuzz testing of the arithmetic operations.	**Moderate**
Auditing	All functions involved in critical state-changing operations emit events. However, it is unclear whether the Increment Finance team has implemented off-chain monitoring or developed an incident response plan. Recommendations on creating an incident response plan are provided in appendix D.	**Moderate**
Authentication / Access Controls	The system uses role-based access controls. The governance and manager roles are responsible for superuser actions, and their responsibilities are documented. However, given the number of privileges provided to the governance role, it constitutes a single point of failure (TOB-INC-1).	**Moderate**
Complexity Management	Because the codebase relies on Curve's vAMM, it must use a complex model to correctly calculate fees. In addition, much of the codebase has high cyclomatic	**Weak**

	complexity, with many different control flows corresponding to each operation; this complexity makes the codebase difficult to test and maintain. The Increment Finance team should review all system flows and identify any functions that could be removed to decrease complexity.	
Decentralization	The system has two privileged actors, both of which are controlled by multisignature wallets; their abilities are documented, but the risks associated with privileged actors should be outlined in greater detail. Additionally, when the protocol is not paused, users can exit the system at will. Finally, the risks related to external contract interactions are also documented. If the Increment Finance team moves from a multisignature architecture to a decentralized autonomous organization (DAO), we recommend that the team thoroughly document the migration plan ahead of that move.	**Moderate**
Documentation	The Increment Finance team has developed thorough user and developer documentation regarding each part of the protocol. It also provided diagrams explaining the end-to-end use of the system, and most functions in the codebase have NatSpec-compliant comments. However, the inline documentation should be expanded to improve the codebase's readability. Additionally, there are some discrepancies between the protocol specification and the implementation.	**Satisfactory**
Front-Running Resistance	We identified two front-running opportunities (TOB-INC-5, TOB-INC-9), both of which could degrade user experience and lead to a DoS. We also identified an arbitrage opportunity, detailed in TOB-INC-4. Users should be provided documentation on this arbitrage opportunity (and any others) and the risk of front-running.	**Weak**

Low-Level Manipulation	Only one function in the codebase uses inline assembly. However, that function should have additional inline documentation outlining the use of assembly and its advantages over a higher-level implementation.	**Satisfactory**
Testing and Verification	While the codebase has high unit test coverage, we identified a number of issues that could have been found with a deeper test suite. We recommend that the Increment Finance team expand its suite of integration tests and implement automated fuzz testing.	**Moderate**

Summary of Findings

The table below summarizes the findings of the review, including type and severity details.

ID	Title	Type	Severity
1	Governance role is a single point of failure	Access Controls	High
2	Inconsistent lower bounds on collateral weights	Data Validation	Medium
3	Solidity compiler optimizations can be problematic	Undefined Behavior	Informational
4	Support for multiple reserve tokens allows for arbitrage	Undefined Behavior	Informational
5	Ownership transfers can be front-run	Timing	High
6	Funding payments are made in the wrong token	Data Validation	High
7	Excessive dust collection may lead to premature closures of long positions	Data Validation	Medium
8	Problematic use of primitive operations on fixed-point integers	Undefined Behavior	Informational
9	Liquidations are vulnerable to sandwich attacks	Timing	Medium
10	Accuracy of market and oracle TWAPs is tied to the frequency of user interactions	Data Validation	Informational
11	Liquidations of short positions may fail because of insufficient dust collection	Data Validation	Low

12	Project dependencies contain vulnerabilities	Patching	**Low**
13	Risks associated with oracle outages	Configuration	**Informational**

Detailed Findings

1. Governance role is a single point of failure	
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.

Exploit Scenario

Eve, an attacker, creates a fake token, compromises the governance account, and adds the fake token as a reserve token for UA. She mints UA by making a deposit of the fake token and then burns the newly acquired UA tokens, which enables her to withdraw all USDC from the reserves.

Recommendations

Short term, minimize the privileges of the governance role and update the documentation to include the implications of those privileges. Additionally, implement reasonable time delays for privileged operations.

Long term, document an incident response plan and ensure that the private keys for the multisig are managed safely. Additionally, carefully evaluate the risks of moving from a multisig to a DAO and consider whether the move is necessary.

2. Inconsistent lower bounds on collateral weights

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.

Exploit Scenario

Alice, who holds the governance role, decides to update the weight of a collateral asset in response to volatile market conditions. By mistake, Alice sets the weight of the collateral to 1% instead of 10%. As a result of this change, Bob's non-UA collateral assets decrease in value and are seized.

Recommendations

Short term, change the lower bound on `newWeight` in the `changeCollateralWeight` function from `1e16` to `1e17`.

Long term, expand the unit test suite to cover additional edge cases and to ensure that the system behaves as expected.

3. Solidity compiler optimizations can be problematic

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.

Exploit Scenario
A latent or future bug in Solidity compiler optimizations causes a security vulnerability in the Increment Protocol contracts.

Recommendations
Short term, measure the gas savings from optimizations and carefully weigh them against the possibility of an optimization-related bug.

Long term, monitor the development and adoption of Solidity compiler optimizations to assess their maturity.

4. Support for multiple reserve tokens allows for arbitrage

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.

Exploit Scenario

Alice, who holds the governance role, adds USDC and DAI as reserve tokens. Eve notices that DAI is trading at USD 0.99, while USDC is trading at USD 1.00. Thus, she decides to mint a large amount of UA by depositing DAI and to subsequently return the DAI and withdraw USDC, allowing her to make a risk-free profit.

Recommendations

Short term, document all front-running and arbitrage opportunities in the protocol to ensure that users are aware of them. As development continues, reassess the risks associated with those opportunities and evaluate whether they could adversely affect the protocol.

Long term, implement an off-chain monitoring solution (like that detailed in TOB-INC-13) to detect any anomalous fluctuations in the prices of supported reserve tokens. Additionally, develop an incident response plan to ensure that any issues that arise can be addressed promptly and without confusion. (See appendix D for additional details on creating an incident response plan.)

5. Ownership transfers can be front-run

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).

Exploit Scenario

Alice, an admin of the Increment Protocol, deploys a new `Perpetual` contract. Alice then attempts to call `transferPerpOwner` to set `perp` to the address of the deployed contract. However, Eve, an attacker monitoring the mempool, sees Alice's call to `transferPerpOwner` and calls the function with a higher gas price. As a result, Eve gains ownership of the virtual tokens and renders the perpetual market useless. Eve then

repeats the process with each subsequent deployment of a perpetual market, executing a DoS attack.

Recommendations

Short term, move all functionality from the `PerpOwnable` contract to the `Perpetual` contract. Then add the `hasRole` modifier to the `transferPerpOwner` function so that the function can be called only by the manager or governance role.

Long term, document all cases in which front-running may be possible, along with the implications of front-running for the codebase.

6. Funding payments are made in the wrong token

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.

Exploit Scenario

The BTC / USD perpetual market's mark price is significantly higher than the index price. Alice, who holds a short position, decides to exit the market. However, the protocol calculates her funding payments in BTC and does not convert them to their UA equivalents before updating her balance. Thus, Alice makes much less than expected.

Recommendations

Short term, use the `vBase.indexPrice()` function to convert vBase token amounts to UA before the call to `vault.settlePnL`.

Long term, expand the unit test suite to cover additional edge cases and to ensure that the system behaves as expected.

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

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.

Exploit Scenario

Alice, who holds a long position in the vBTC / vUSD market, decides to close most of her position. After the swap, `netPositionSize` is slightly less than 1e17. Since a leftover balance of that amount is considered dust (unbeknownst to Alice), her ~1e17 vBTC tokens are sent to the `Insurance` contract, and her position is fully closed.

Recommendations

Short term, have the protocol calculate the notional value of `netPositionSize` by multiplying it by the return value of the `indexPrice` function. Then have it compare that notional value to the dust thresholds. Note that the dust thresholds must also be expressed in the notional token and that the comparison should not lead to a significant decrease in a user's position.

Long term, document this system edge case to inform users that a fraction of their long positions may be donated to the `Insurance` contract after being reduced.

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

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.

Recommendations

Short term, identify all state variables that are fixed-point signed or unsigned integers. Additionally, ensure that all multiplication and division operations involving those state variables use the `wadMul` and `wadDiv` functions, respectively. If the Increment Finance team decides against using `wadMul` or `wadDiv` in any of those operations (whether to optimize gas or for another reason), it should provide inline documentation explaining that decision.

9. Liquidations are vulnerable to sandwich attacks

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.

Exploit Scenario

Alice, a liquidator, notices that a position is no longer valid and decides to liquidate it. When she sends the transaction, the protocol sets the minimum-amount-out value to zero. Eve's sandwich bot identifies Alice's liquidation as a pure profit opportunity and sandwiches it with transactions. Alice's liquidation fails, and the protocol remains in a state of debt.

Recommendations

Short term, allow liquidators to specify a minimum-amount-out value when liquidating the positions of traders and liquidity providers.

Long term, document all cases in which front-running may be possible, along with the implications of front-running for the codebase.

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

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.

Recommendations

Short term, create a public `updateGlobalState` function that anyone can call to internally call `_updateGlobalState`.

Long term, create an off-chain worker that can alert the team to periods of perpetual market inactivity, ensuring that the team knows to update the market accordingly.

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

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.

Recommendations

Short term, take the following steps:

1. Implement the short-term recommendation outlined in TOB-INC-7 to prevent the collection of an excessive amount of dust.
2. When the protocol is liquidating a short position, take the absolute value of `netPositionSize` and check whether it can be considered dust. If it can, zero out the position's balance, but do not donate the position's balance to the `Insurance` contract. A non-zero `netPositionSize` for a short position means that the position holds a debt, and that debt should not be transferred to insurance.
3. Remove the checks of `netBasePosition` from the `_settleLpPosition` function. (The changes made in the first two steps will render them redundant.)
4. Add a check of the `_isTraderPositionOpen` function's return value at the end of the `_liquidateLp` function to ensure that the account's `openNotional` and `positionSize` values are equal to zero.

Long term, implement the long-term recommendation outlined in TOB-INC-7. Additionally, document the fact that a liquidator should use the `CurveCryptoViews.get_dy_ex_fees` function to obtain an accurate estimate of the `proposedAmount` value before attempting to close a short position.

12. Project dependencies contain vulnerabilities

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

Exploit Scenario

Alice installs the dependencies of the in-scope repository on a clean machine. Unbeknownst to Alice, a dependency of the project has become malicious or exploitable. Alice subsequently uses the dependency, disclosing sensitive information to an unknown actor.

Recommendations

Short term, ensure that the Increment Protocol dependencies are up to date. Several node modules have been documented as malicious because they execute malicious code when installing dependencies to projects. Keep modules current and verify their integrity after installation.

Long term, integrate automated dependency auditing into the development workflow. If a dependency cannot be updated when a vulnerability is disclosed, ensure that the code does not use and is not affected by the vulnerable functionality of the dependency.

13. Risks associated with oracle outages

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

References

- Chainlink: Risk Mitigation
- Chainlink: Monitoring Data Feeds
- Chainlink: Circuit Breakers

Summary of Recommendations

The Increment Protocol is a work in progress with multiple planned iterations. Trail of Bits recommends that Increment Finance address the findings detailed in this report and take the following additional steps prior to deployment:

- Identify and analyze all system properties that are expected to hold.
- Use Echidna to test and validate those system properties.
- Develop a detailed incident response plan to ensure that any issues that arise can be addressed promptly and without confusion. (See appendix D for related recommendations.)
- Ensure that all potential front-running, sandwiching, and arbitrage opportunities are either mitigated or thoroughly documented.
- Ensure that all fixed-point arithmetic is performed correctly and with the provided library operations.

A. 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.

B. Code Maturity Categories

The following tables describe the code maturity categories and rating criteria used in this document.

Code Maturity Categories	
Category	**Description**
Arithmetic	The proper use of mathematical operations and semantics
Auditing	The use of event auditing and logging to support monitoring
Authentication / Access Controls	The use of robust access controls to handle identification and authorization and to ensure safe interactions with the system
Complexity Management	The presence of clear structures designed to manage system complexity, including the separation of system logic into clearly defined functions
Cryptography and Key Management	The safe use of cryptographic primitives and functions, along with the presence of robust mechanisms for key generation and distribution
Decentralization	The presence of a decentralized governance structure for mitigating insider threats and managing risks posed by contract upgrades
Documentation	The presence of comprehensive and readable codebase documentation
Front-Running Resistance	The system's resistance to front-running attacks
Low-Level Manipulation	The justified use of inline assembly and low-level calls
Testing and Verification	The presence of robust testing procedures (e.g., unit tests, integration tests, and verification methods) and sufficient test coverage

Rating Criteria	
Rating	**Description**
Strong	No issues were found, and the system exceeds industry standards.
Satisfactory	Minor issues were found, but the system is compliant with best practices.
Moderate	Some issues that may affect system safety were found.
Weak	Many issues that affect system safety were found.
Missing	A required component is missing, significantly affecting system safety.
Not Applicable	The category is not applicable to this review.
Not Considered	The category was not considered in this review.
Further Investigation Required	Further investigation is required to reach a meaningful conclusion.

C. Multisignature Wallet Best Practices

Consensus requirements for sensitive actions such as spending the funds in a wallet are meant to mitigate the risk of

- any one person's judgment overruling the others',
- any one person's mistake causing a failure, and
- the compromise of any one person's credentials causing a failure.

In a 2-of-3 multisignature Ethereum wallet, for example, the execution of a "spend" transaction requires the consensus of two individuals in possession of two of the wallet's three private keys. For this model to be useful, it must fulfill the following requirements:

1. The private keys must be stored or held separately, and access to each one must be limited to a different individual.
2. If the keys are physically held by third-party custodians (e.g., a bank), multiple keys should not be stored with the same custodian. (Doing so would violate requirement #1.)
3. The person asked to provide the second and final signature on a transaction (i.e., the co-signer) ought to refer to a pre-established policy specifying the conditions for approving the transaction by signing it with his or her key.
4. The co-signer also ought to verify that the half-signed transaction was generated willfully by the intended holder of the first signature's key.

Requirement #3 prevents the co-signer from becoming merely a "deputy" acting on behalf of the first signer (forfeiting the decision-making responsibility to the first signer and defeating the security model). If the co-signer can refuse to approve the transaction for any reason, the due-diligence conditions for approval may be unclear. That is why a policy for validating transactions is needed. A verification policy could include the following:

- A protocol for handling a request to co-sign a transaction (e.g., a half-signed transaction will be accepted only via an approved channel)
- A whitelist of specific Ethereum addresses allowed to be the payee of a transaction
- A limit on the amount of funds spent in a single transaction, or in a single day

Requirement #4 mitigates the risks associated with a single stolen key. For example, say that an attacker somehow acquired the unlocked Ledger Nano S of one of the signatories. A voice call from the co-signer to the initiating signatory to confirm the transaction would reveal that the key had been stolen and that the transaction should not be co-signed. If the signatory were under an active threat of violence, he or she could use a "duress code" (a code word, a phrase, or another signal agreed upon in advance) to covertly alert the others that the transaction had not been initiated willfully, without alerting the attacker.

D. Incident Response Plan Recommendations

This section provides recommendations on formulating an incident response plan.

- **Identify the parties (either specific people or roles) responsible for implementing the mitigations when an issue occurs (e.g., deploying smart contracts, pausing contracts, upgrading the front end, etc.).**
- **Document internal processes for addressing situations in which a deployed remedy does not work or introduces a new bug.**
    - Consider documenting a plan of action for handling failed remediations.
- **Clearly describe the intended contract deployment process.**
- **Outline the circumstances under which Increment Finance will compensate users affected by an issue (if any).**
    - Issues that warrant compensation could include an individual or aggregate loss or a loss resulting from user error, a contract flaw, or a third-party contract flaw.
- **Document how the team plans to stay up to date on new issues that could affect the system; awareness of such issues will inform future development work and help the team secure the deployment toolchain and the external on-chain and off-chain services that the system relies on.**
    - Identify sources of vulnerability news for each language and component used in the system, and subscribe to updates from each source. Consider creating a private Discord channel in which a bot will post the latest vulnerability news; this will provide the team with a way to track all updates in one place. Lastly, consider assigning certain team members to track news about vulnerabilities in specific components of the system.
- **Determine when the team will seek assistance from external parties (e.g., auditors, affected users, other protocol developers, etc.) and how it will onboard them.**
    - Effective remediation of certain issues may require collaboration with external parties.

- **Define contract behavior that would be considered abnormal by off-chain monitoring solutions.**

It is best practice to perform periodic dry runs of scenarios outlined in the incident response plan to find omissions and opportunities for improvement and to develop "muscle memory." Additionally, document the frequency with which the team should perform dry runs of various scenarios, and perform dry runs of more likely scenarios more regularly. Create a template to be filled out with descriptions of any necessary improvements after each dry run.

E. Code Quality Recommendations

The following recommendations are not associated with specific vulnerabilities. However, they enhance code readability and may prevent the introduction of vulnerabilities in the future.

- **Replace the conditional expression of form highlighted in figure E.1 with the equivalent expression highlighted in figure E.2.**

```
uint256 tokenIdx = tokenToCollateralIdx[withdrawToken];
if (!((tokenIdx != 0) || (address(withdrawToken) == address(UA)))) revert
Vault_UnsupportedCollateral();
```

Figure E.1: The conditional expression used to verify the validity of a collateral token in Vault.sol#L86-L87

```
uint256 tokenIdx = tokenToCollateralIdx[withdrawToken];
if ((tokenIdx == 0) && (address(withdrawToken) != address(UA))) revert
Vault_UnsupportedCollateral();
```

Figure E.2: The proposed improvement to the code in figure E.1

- **Update the code documentation so that it accurately describes the implementation.**

  The code documentation highlighted in figure E.3 indicates that balance amounts may not have 18 decimals. However, balance amounts are converted to 18-decimal precision before they are added to the mapping.

```
//       user  =>    collateralIdx => balance (might not be 18 decimals)
mapping(address => mapping(uint256 => int256)) private balances;
```

Figure E.3: An inaccuracy in Vault.sol#L45-L46

  The code documentation in figure E.4 is not applicable to the current implementation, since the vulnerability is no longer exploitable and there is no `_checkProposedAmount()` function in the codebase.

```
/*
...
 @audit Note that this mechanism can be exploited by inserting a large value here,
since traders
        will encounter slippage on the curve trading pool. We set a limit of 1.5 x
market value in _checkProposedAmount()
*/
```

Figure E.4: An inapplicable code comment in the `Perpetual` contract (`Perpetual.sol#L843-844`)

- **Rename the `tokenAmount` variable to `wadAmount`.** The latter name will make it clear that the amount is converted to 18-decimal precision before the variable's assignment.

```
uint256 tokenAmount =
LibReserve.tokenToWad(whiteListedCollaterals[tokenIdx].decimals, amount);
```

Figure E.5: A snippet of the `withdraw` function in `Vault.sol#L108`

- **Remove the `tokenIdx` check from `Vault._withdraw`.** The `tokenIdx` value is already checked in `withdraw` and `withdrawAll`.

```
uint256 tokenIdx = tokenToCollateralIdx[withdrawToken];
if (!((tokenIdx != 0) || (address(withdrawToken) == address(UA)))) revert
Vault_UnsupportedCollateral();
```

Figure E.6: A snippet of the `_withdraw` function in `Vault.sol#L367-368`

- **Update the bounds of the `insuranceFee` parameter in the following conditional expression to reflect the documentation.** The documentation indicates that the valid range for insurance fees is [0.001%, 0.1%], which is equal to [1e13, 1e14], not [1e14, 1e16].

```
if (params.insuranceFee < 1e14 || params.insuranceFee > 1e16)
    revert Perpetual_InsuranceFeeInvalid(params.insuranceFee);
```

Figure E.7: A snippet of the `setParameters` function in `Perpetual.sol#L467-468`