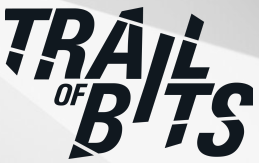

Perpetual Protocol V2

Security Assessment

March 22, 2022

Prepared for:

Perpetual Finance

Prepared by: **Michael Colburn, Paweł Płatek, and Maciej Domański**

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 mutually 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

Perpetual Finance engaged Trail of Bits to review the security of its Perpetual Protocol V2 smart contracts. From February 14 to March 4, 2022, a team of three consultants conducted a security review of the client-provided source code, with six 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 documentation. We performed static testing of the target system, using both automated and manual processes.

Summary of Findings

The audit did not uncover any significant flaws or defects that could impact system confidentiality, integrity, or availability. A summary of the findings is provided below.

EXPOSURE ANALYSIS

Severity	*Count*
High	0
Medium	4
Low	0
Informational	5
Undetermined	0

CATEGORY BREAKDOWN

Category	*Count*
Data Validation	6
Undefined Behavior	3

Project Summary

Contact Information

The following managers were associated with this project:

Dan Guido, Account Manager
dan@trailofbits.com

Sam Greenup, Project Manager
sam.greenup@trailofbits.com

The following engineers were associated with this project:

Michael Colburn, Consultant
michael.colburn@trailofbits.com

Paweł Płatek, Consultant
pawel.platek@trailofbits.com

Maciej Domański, Consultant
maciej.domanski@trailofbits.com

Project Timeline

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

Date	Event
February 10, 2022	Pre-project kickoff call
February 18, 2022	Status update meeting #1
February 25, 2022	Status update meeting #2
March 4, 2022	Delivery of report draft
March 4, 2022	Report readout meeting
March 14, 2022	Delivery of final report

Project Goals

The engagement was scoped to provide a security assessment of the Perpetual Protocol V2 smart contracts. Specifically, we sought to answer the following non-exhaustive list of questions:

- Are there any flaws or inconsistencies in the internal accounting?
- Are the various fees applied correctly?
- Does the system's modular architecture introduce any classes of bugs?
- Can tokens be moved from the vault outside of the expected flows?
- Is it possible to trigger a liquidation outside of the intended parameters?
- Do the contracts perform appropriate input validation and access control checks?

Project Targets

The engagement involved a review and testing of the targets listed below.

perp-lushan

Repository	https://github.com/perpetual-protocol/perp-lushan
Version	`bac1ae0b6dd633275b175e06169c5cb02896b8e5`
Type	Solidity
Platform	Ethereum

perp-oracle

Repository	https://github.com/perpetual-protocol/perp-oracle
Version	`ba78a5b87098dcffb7285fc585afff1001a87232`
Type	Solidity
Platform	Ethereum

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`.** The `ClearingHouse` contract is the main entry point of the protocol for users. It allows users to manage liquidity, open and close long or short positions, and perform liquidations of underwater positions. We reviewed this contract to ensure that it correctly updates liquidity in the protocol, that it properly modifies long and short positions, and that the interactions with the other core contracts are sound.

- **`Exchange` and `OrderBook`.** These contracts function as wrappers for performing swaps and managing liquidity in the Uniswap V3 pools for the protocol's virtual tokens. We reviewed these contracts to ensure that they properly interact with Uniswap, that orders are tracked consistently internally, and that fee amounts are calculated correctly.

- **`AccountBalance`.** This contract serves as a ledger to track the internal accounting for the protocol. We reviewed the contract to ensure that all of the bookkeeping is consistent and that the calculations are sound.

- **`Vault`.** The `Vault` contract stores user collateral. We reviewed this contract to ensure that tokens are handled properly within it and that they cannot be accessed without permission.

- **Oracles.** The oracle contracts serve as wrappers to external price feeds for the Perpetual Protocol system. We reviewed the various price feed contracts to ensure that they properly interact with the external systems and cannot be manipulated.

Coverage Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. During this project, we were unable to perform comprehensive testing of the following system elements, which may warrant further review:

- Automated testing of the protocol with Echidna

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 contracts use safe arithmetic libraries to prevent overflows and to ensure safe casting between integer types. Any instances that do not use these libraries either cannot overflow or have comments indicating the intended behavior.	**Satisfactory**
Auditing	Many functions in the contracts emit events when necessary. In addition to this measure, we encourage the Perpetual Finance team to develop an incident response plan and implement on-chain monitoring (if not already implemented).	**Satisfactory**
Authentication / Access Controls	There are appropriate access controls in place for privileged operations, both for operations between contracts and for operations regarding contract ownership.	**Satisfactory**
Complexity Management	The functions and contracts are organized and scoped appropriately and contain inline documentation that explains their workings. Though there is a clear separation of duties between each of the core contracts in the system, this can cause some interactions to be difficult to follow as they pass between different contracts.	**Satisfactory**
Cryptography and Key Management	This category was not in scope for this assessment.	**Not Considered**

Decentralization	The contracts have several parameters that can be updated by the contract owner after deployment. Additionally, the contracts are deployed behind proxies, which allows the implementations to be upgraded in the future. Ownership of the contracts is transferred to a Gnosis multisignature wallet at the end of the deployment process.	**Satisfactory**
Documentation	The project has good high-level documentation, a specification indicating the derivations for the formulas used by the system, and good use of NatSpec and inline comments.	**Satisfactory**
Front-Running Resistance	The Perpetual Finance team informed us of front-running issues that they were already aware of. We did not identify any other front-running issues during this review, but this category requires further investigation.	**Further Investigation Required**
Low-Level Calls	The contracts do not use assembly or make low-level calls aside from their use of the `delegatecall` proxy pattern.	**Not Applicable**
Testing and Verification	The codebase includes test cases for a wide variety of scenarios, though we could not determine the exact coverage rate. There are several "TODO" comments throughout the test files that should be addressed. We also suggest looking into augmenting the test suite with automated testing.	**Moderate**

Summary of Findings

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

ID	Title	Type	Severity
1	Lack of zero-value checks on functions	Data Validation	Informational
2	Solidity compiler optimizations can be problematic	Undefined Behavior	Informational
3	mulDiv reverts instead of returning MIN_INT	Data Validation	Informational
4	Discrepancies between code and specification	Undefined Behavior	Informational
5	Missing Chainlink price feed safety checks	Data Validation	Medium
6	Band price feed may return invalid prices in two edge cases	Data Validation	Medium
7	Ever-increasing priceCumulative variables	Undefined Behavior	Medium
8	Lack of rounding in Emergency price feed	Data Validation	Informational
9	It is possible to pollute the observations array	Data Validation	Medium

Detailed Findings

1. Lack of zero-value checks on functions	
Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-PERP-1
Target: `ClearingHouseCallee.sol`, `OrderBook.sol`	

Description

The `ClearingHouseCallee` contract's `setClearingHouse` function and the `OrderBook` contract's `setExchange` function fail to validate some of their incoming arguments, so callers can accidentally set important state variables to the zero address.

```
function setClearingHouse(address clearingHouseArg) external onlyOwner {
    _clearingHouse = clearingHouseArg;
    emit ClearingHouseChanged(clearingHouseArg);
}
```

Figure 1.1: Missing zero-value check (perp-lushan/contracts/base/ClearingHouseCallee.sol#30–33)

```
function setExchange(address exchangeArg) external onlyOwner {
    _exchange = exchangeArg;
    emit ExchangeChanged(exchangeArg);
}
```

Figure 1.2: Missing zero-value check (perp-lushan/contracts/OrderBook.sol#93–96)

Exploit Scenario

Alice, a Perpetual Finance team member, mistakenly provides the zero address as an argument when configuring a `ClearingHouseCallee` contract. As a result, the `_clearingHouse` for this instance is set to the zero address instead of the intended `ClearingHouse`, and some contract functionality fails unexpectedly until it is reconfigured.

Recommendations

Short term, add zero-value or other contract existence checks for all function arguments to ensure that users cannot mistakenly set incorrect values, misconfiguring the system.

Long term, use Slither, which will catch functions that do not have zero-value checks.

2. Solidity compiler optimizations can be problematic	
Severity: **Informational**	Difficulty: **High**
Type: Undefined Behavior	Finding ID: TOB-PERP-2
Target: `perp-lushan/hardhat.config.ts`, `perp-oracle/hardhat-config.ts`	

Description
The Perpetual Protocol V2 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.

High-severity security issues due to optimization bugs have occurred in the past. A high-severity bug in the `emscripten`-generated `solc-js` compiler used by Truffle and Remix persisted until late 2018. The fix for this bug was not reported in the Solidity CHANGELOG. Another high-severity optimization bug resulting in incorrect bit shift results was patched in Solidity 0.5.6. More recently, another bug due to the incorrect caching of `keccak256` was reported.

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—or in the Emscripten transpilation to `solc-js`—causes a security vulnerability in the Perpetual Protocol V2 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.

3. mulDiv reverts instead of returning MIN_INT

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-PERP-3
Target: `PerpMath.sol`	

Description

The `mulDiv` function cannot return the minimum signed value (-2^{255}); it reverts on receiving this value.

The function takes as arguments signed numbers, internally operates on the unsigned type, and then converts the result to the signed type, as shown in figure 3.1.

```
result = negative ? neg256(unsignedResult) : PerpSafeCast.toInt256(unsignedResult);
```

Figure 3.1: Final conversion in `mulDiv` (perp-lushan/contracts/lib/PerpMath.sol#84)

The `neg256` function converts a number to the signed type and negates it. The `toInt256` method checks whether the number is less or equal to the maximal signed value ($2^{255} - 1$). So if the unsigned result passed to `neg256` is 2^{255}, then the function will revert.

```
function neg256(uint256 a) internal pure returns (int256) {
    return -PerpSafeCast.toInt256(a);
}
```

Figure 3.2: Casting to `int256` (perp-lushan/contracts/lib/PerpMath.sol#45–47)

```
function toInt256(uint256 value) internal pure returns (int256) {
    require(value <= uint256(type(int256).max), "SafeCast: value doesn't fit in an
int256");
    return int256(value);
}
```

Figure 3.3: The check that incorrectly fails for `type(int256).max` value (perp-lushan/contracts/lib/PerpSafeCast.sol#183–186)

Recommendations

Short term, ensure that this behavior is documented and will not cause unexpected reverts.

4. Discrepancies between code and specification	
Severity: **Informational**	Difficulty: **High**
Type: Undefined Behavior	Finding ID: TOB-PERP-4
Target: `ClearingHouse.sol`	

Description

While reviewing the Perpetual Protocol contracts, we compared the implementation against the provided specification. We noted some minor discrepancies between the two.

In the "Account Specs" section of the specification, account value is calculated in the following way:

```
accountValue = collateral + owedRealizedPnl + pendingFundingPayment +
                            pendingFee + unrealizedPnl
```

However, in `ClearingHouse.getAccountValue`, the `pendingfundingPayment` amount is subtracted instead of added.

In the "Liquidation" section of the specification, the liquidation fee is calculated in the following way:

```
liquidationFee = exchangePositionNotional * liquidationPenaltyRatio
```

However, in `ClearingHouse._liquidate`, `liquidationFee` uses the absolute value of `exchangedPositionNotional` instead.

Recommendations

Short term, update or clarify the specification to match the implementation, or vice versa.

Long term, regularly review the specification and corresponding implementation to ensure they accurately reflect the system as designed.

5. Missing Chainlink price feed safety checks

Severity: **Medium**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-PERP-5
Target: `ChainlinkPriceFeed.sol`	

Description

Certain safety checks that should be used to validate data returned from `latestRoundData` and `getRoundData` are missing:

- `require(updatedAt > 0)`: This checks whether the requested round is valid and complete; an example use of the check can be found in the `historical-price-feed-data` project, and a description of the parameter validation can be found in Chainlink's documentation.
- `latestPrice > 0`: While price is expected to be greater than zero, the code may consume zero prices, as shown in figures 5.1 and 5.2. This check should be added before all `return` calls.

```
if (interval == 0 || round == 0 || latestTimestamp <= baseTimestamp) {
    return latestPrice;
}
```

Figure 5.1: The code could return a price of zero. (perp-oracle/contracts/ChainlinkPriceFeed.sol#47–49)

```
if (latestPrice < 0) {
    _requireEnoughHistory(round);
    (round, finalPrice, latestTimestamp) = _getRoundData(round - 1);
}
return (round, finalPrice, latestTimestamp);
```

Figure 5.2: The code could return a price of zero. (perp-oracle/contracts/ChainlinkPriceFeed.sol#95–99)

- `require(answeredInRound == roundId)`: As the documentation specifies, "If `answeredInRound` is less than `roundId`, the answer is being carried over. If `answeredInRound` is equal to `roundId`, then the answer is fresh."

- `require(latestTimestamp > baseTimestamp)`: Currently, the oracle may return a very outdated price, as shown in figure 5.1. The code should revert if no fresh price is available.

Also note that, according to the documentation, `roundId` "increases with each new round," but the "increase might not be monotonic" (probably meaning that `roundId` does not increase by one). Currently, the code iterates backward over `round` values one by one, which may be incorrect. There should be checks for returned `updatedAts`:

- If `updatedAt` values are decreasing (e.g., as shown in the `historical-price-feed-data` project)
- If `updatedAt` values are zero, if expected
    - `updatedAt` may be equal to zero for missing rounds, which may indicate that the requested `roundId` is invalid. For example, see these checks in the `historical-price-feed-data` project. Please note that requesting invalid round details may revert instead of returning empty data.

Exploit Scenario

Because of a bug in Chainlink, `latestRoundData` returns uninitialized data. The `round` variable equals zero, so zero is returned as the price. The bug is noticed by an attacker who uses it to drain the protocol.

Recommendations

Short term, implement checks for the scenarios described above.

Long term, add a requirement for a minimum number of calls to the `_getRoundData` method to ensure that enough data is used to compute `weightedPrice`. Add tests for the Chainlink price feed with various edge cases, possibly with specially crafted random data. Review the specific implementation of Chainlink Aggregator that the system will use and adjust the system to its semantic.

6. Band price feed may return invalid prices in two edge cases

Severity: **Medium**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-PERP-6
Target: `BandPriceFeed.sol`	

Description

The Band price feed returns a price instead of reverting in the following two edge cases: 1) there is not enough historical data and an entry in the `observations` array is empty, and 2) the historical data is very old. The prices returned in the context of these edge cases could be incorrect.

As shown in figure 6.1, if there is not enough data, the code reverts. However, if at the same time an `observations` entry is empty, the code will not revert, as shown in figure 6.2, but will continue executing the code, taking the branch in figure 6.3.

```
// not enough historical data to query
if (i == observationLen) {
    // BPF_NEH: no enough historical data
    revert("BPF_NEH");
}
```

Figure 6.1: The code reverts if there is not enough historical data. (perp-oracle/contracts/BandPriceFeed.sol#220–224)

```
// if the next observation is empty, using the last one
// it implies the historical data is not enough
if (observations[index].timestamp == 0) {
    atOrAfterIndex = beforeOrAtIndex = index + 1;
    break;
}
```

Figure 6.2: There is not enough historical data, but the condition in figure 6.1 will not be met. (perp-oracle/contracts/BandPriceFeed.sol#207–212)

```
// case1. not enough historical data or just enough (`==` case)
if (targetTimestamp <= beforeOrAt.timestamp) {
    targetTimestamp = beforeOrAt.timestamp;
    targetPriceCumulative = beforeOrAt.priceCumulative;
}
```

Figure 6.3: This branch executes if there is not enough historical data and not enough observations. (perp-oracle/contracts/BandPriceFeed.sol#119–123)

So, the price may be computed with a shorter time period (interval) than the user expected. This indicates that it is easier than expected to manipulate the price of a newly added token.

The second edge case is when the historical data is very old. In this case, the code will compute the price using the oldest observation and the recently requested data, as shown in figures 6.4 and 6.5.

```
uint256 currentPriceCumulative =
    lastestObservation.priceCumulative +
        (lastestObservation.price * (latestBandData.lastUpdatedBase -
lastestObservation.timestamp)) +
        (latestBandData.rate * (currentTimestamp - latestBandData.lastUpdatedBase));

[redacted]

// case2. the latest data is older than or equal the request
else if (atOrAfter.timestamp <= targetTimestamp) {
    targetTimestamp = atOrAfter.timestamp;
    targetPriceCumulative = atOrAfter.priceCumulative;
}

[redacted]

return (currentPriceCumulative - targetPriceCumulative) / (currentTimestamp -
targetTimestamp);
```

Figure 6.4: Computation of the price with old data (perp-oracle/contracts/BandPriceFeed.sol#105–139)

Because `atOrAfter` is the latest entry in the `observations` array (`lastestObservation`), we can reduce the equation in the following way:

```
price * (currentTimestamp - targetTimestamp) =

 currentPriceCumulative - targetPriceCumulative =

 lastestObservation.priceCumulative +
 (lastestObservation.price * (latestBandData.lastUpdatedBase - lastestObservation.timestamp)) +
 (latestBandData.rate * (currentTimestamp - latestBandData.lastUpdatedBase)) -
 atOrAfter.priceCumulative =

 (lastestObservation.price * (latestBandData.lastUpdatedBase - lastestObservation.timestamp)) +
 (latestBandData.rate * (currentTimestamp - latestBandData.lastUpdatedBase))
```

Figure 6.5: Reduced equations in the case of old data

The result of `currentTimestamp - latestBandData.lastUpdatedBase` should be small, and the result of `latestBandData.lastUpdatedBase - lastestObservation.timestamp` can be large. So the final price will be determined by the oldest observation—the `lastestObservation.price` variable. Moreover, if the last update of the Band price was made in the same block as the call to `getPrice`, then we have `currentTimestamp - latestBandData.lastUpdatedBase == 0`; therefore, the returned price will equal the outdated `lastestObservation.price`.

Exploit Scenario

The Band price feed is not updated for a long time for a certain token. An attacker observes this fact and drains the protocol using the wrongly priced token.

Recommendations

Short term, take the following actions:

- Modify the code so that it always reverts if it does not receive enough historical data, even if there are not yet enough observations. Change the condition in figure 6.3 so that it handles only cases of equality and reverts if `targetTimestamp` is less than `beforeOrAt.timestamp`.
- Modify the code so that it always reverts if the latest data is older than requested and executes only when the latest data is equal to the requested data.

Long term, add tests for the Band price feed with various edge cases, possibly with specially crafted random data.

7. Ever-increasing priceCumulative variables

Severity: **Medium**	Difficulty: **High**
Type: Undefined Behavior	Finding ID: TOB-PERP-7
Target: `BandPriceFeed.sol`	

Description

In the Band price feed, `observations.priceCumulative` variables can increase indefinitely and overflow. Every call to the `update` method adds to one of the `observations.priceCumulative` variables, and there is neither a check for overflows nor a way to reset the variable (or the whole `observations` array).

```
uint256 elapsedTime = bandData.lastUpdatedBase - lastObservation.timestamp;
observations[currentObservationIndex] = Observation({
    priceCumulative: lastObservation.priceCumulative + (lastObservation.price *
elapsedTime),
    timestamp: bandData.lastUpdatedBase,
    price: bandData.rate
});
```

Figure 7.1: Part of the `update` method (`perp-oracle/contracts/BandPriceFeed.sol#76–81`)

Exploit Scenario

For a short period of time, Band reports a very high, incorrect price for a certain token. The time-weighted average price (TWAP) mechanism reduces the impact of this bug; however, `priceCumulative` variables become large. After some time, they overflow silently, breaking internal invariants. The Band price feed starts returning high, incorrect prices for the token. An attacker exploits this issue to drain the protocol.

Recommendations

Short term, add a check for overflows of `priceCumulative` variables. If an overflow is detected, the code should disable the price feed or handle the overflow correctly.

8. Lack of rounding in Emergency price feed

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-PERP-8
Target: `EmergencyPriceFeed.sol`	

Description

The Emergency price feed does not round down negative arithmetic mean ticks, as the Uniswap's `OracleLibrary` does. Computations done by the Emergency price feed are shown in figure 8.1.

```
// tick(imprecise as it's an integer) to price
return TickMath.getSqrtRatioAtTick(int24((tickCumulatives[1] - tickCumulatives[0]) /
twapInterval));
```

Figure 8.1: Emergency price feed computations of arithmetic mean ticks (perp-oracle/contracts/EmergencyPriceFeed.sol#65–66)

Computations performed in the `OracleLibrary`'s `consult` method are shown in figure 8.2.

```
arithmeticMeanTick = int24(tickCumulativesDelta / secondsAgo);
// Always round to negative infinity
if (tickCumulativesDelta < 0 && (tickCumulativesDelta % secondsAgo != 0))
arithmeticMeanTick--;
```

Figure 8.2: Uniswap computations of arithmetic mean ticks (v3-periphery/contracts/libraries/OracleLibrary.sol#34–36)

Recommendations

Short term, consider modifying the Emergency price feed so that it rounds down negative arithmetic mean ticks before calling the `getSqrtRatioAtTick` method.

9. It is possible to pollute the observations array	
Severity: **Medium**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-PERP-9
Target: `CumulativeTwap.sol`	

Description

In new versions of the Chainlink and Band price feeds, it is possible to pollute the `observations` array with a single observation because the strict equality in `require`'s condition was changed to a loose equality, as shown in figure 9.1.

```
// add `==` in the require statement in case that two or more price with the same
timestamp
// this might happen on Optimism bcs their timestamp is not up-to-date
Observation memory lastObservation = observations[currentObservationIndex];
require(lastUpdatedTimestamp >= lastObservation.timestamp, "CT_IT");
```

Figure 9.1: The insecure `require` condition in the `_update` method in the new version of the code (perp-oracle/contracts/CumulativeTwap.sol#43–46)

Exploit Scenario

An attacker calls the `update` method 256 times in a single transaction. The price feeds do not have historical data and either use only a single price for the TWAP (making price manipulation easy) or stop working.

Recommendations

Short term, use a strict inequality in the `require` statement. Research how to handle Optimism timestamps securely. Alternatively, require a strictly greater timestamp or equal timestamp and a different price; this should prevent attackers from polluting the `observations` array with the same data and should work correctly in Optimism. Make sure to review and mitigate risks introduced by such changes.

Summary of Recommendations

Trail of Bits recommends that Perpetual Finance address the findings detailed in this report and take the following additional steps prior to deployment:

- Continue to develop the test suite, including the areas marked with “TODO” comments. Integrate automated testing into the development workflow.
- Integrate Slither into the CI process.
- Regularly review and update the documentation to ensure it matches the current codebase.
- If not already in place, set up a blockchain monitoring system to detect issues with oracles, potentially malicious liquidations, and similar scenarios.
- Develop an incident response plan to address the above scenarios and other scenarios that may necessitate an emergency response from the Perpetual Finance team.

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 Calls	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. Code Quality Recommendations

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

General

- Address outstanding “@audit” and “TODO” notes in contract comments or add them to the issue tracker.

`perp-oracle/BandPriceFeed`

- There is a typo in the `lastestObservation` variable in the `getPrice` function. It should be updated to `latestObservation`.
- The code relies on overflows to work correctly. If the Solidity compiler is updated to version 0.8.0 or higher, the code will break. Figure C.1 shows an example of where the code relies on overflows.

```
// overflow of currentObservationIndex is desired since currentObservationIndex is
uint8 (0 - 255),
// so 255 + 1 will be 0
currentObservationIndex++;
```

Figure C.1: Expected overflows (`contracts/BandPriceFeed.sol#72–74`)

`perp-oracle/EmergencyPriceFeed`

- The `pool` variable could be declared as immutable as a minor gas optimization measure.

`perp-lushan`

- Consider replacing `1e6` in the `checkRatio` modifier in `ClearingHouseConfig` and `MarketRegistry` as well as `18` in the `initialize` function in `Vault` with named constants to improve readability.

D. Preliminary System Properties

While reviewing the codebase, we identified initial system properties that could be tested using Echidna, our smart contract fuzzer. Due to time constraints, we did not have the opportunity to write the tests, but we encourage the Perpetual Finance team to consider doing so going forward.

`VirtualToken`

- Addresses that are not allowlisted cannot send tokens.
- `totalySupply` is fixed at `uint256(max)`.
- Only the contract owner can add or remove users from the allowlist.

`BaseToken`

- Only the contract owner can pause the token.
- Only the contract owner can change the price feed.
- The token can always be closed after `MAX_WAITING_PERIOD`.
- Once paused or closed, the token cannot be returned to the open state.

`OrderBook`

- `updateFundingGrowthAndLiquidityCoefficientInFundingPayment` and `getLiquidityCoefficientInFundingPayment` calculate the same values outside of state updates.

`Exchange`

- `getPendingFundingPayment` and `_updateFundingGrowth` calculate the same values outside of state updates.

`AccountBalance`

- `getTotalAbsPositionValue` is greater than or equal to `getTotalPositionValue`.

`InsuranceFund`

- The contract's balance does not decrease if a `borrower` has not been set.

E. Token Integration Checklist

The following checklist provides recommendations for interactions with arbitrary tokens. Every unchecked item should be justified, and its associated risks, understood. For an up-to-date version of the checklist, see `crytic/building-secure-contracts`.

For convenience, all Slither utilities can be run directly on a token address, such as the following:

```
slither-check-erc 0xdac17f958d2ee523a2206206994597c13d831ec7 TetherToken --erc erc20
slither-check-erc 0x06012c8cf97BEaD5deAe237070F9587f8E7A266d KittyCore --erc erc721
```

To follow this checklist, use the below output from Slither for the token:

```
slither-check-erc [target] [contractName] [optional: --erc ERC_NUMBER]
slither [target] --print human-summary
slither [target] --print contract-summary
slither-prop . --contract ContractName # requires configuration, and use of Echidna
and Manticore
```

General Considerations

- ❏ **The contract has a security review.** Avoid interacting with contracts that lack a security review. Check the length of the assessment (i.e., the level of effort), the reputation of the security firm, and the number and severity of the findings.
- ❏ **You have contacted the developers.** You may need to alert their team to an incident. Look for appropriate contacts on `blockchain-security-contacts`.
- ❏ **They have a security mailing list for critical announcements.** Their team should advise users (like you!) when critical issues are found or when upgrades occur.

Contract Composition

- ❏ **The contract avoids unnecessary complexity.** The token should be a simple contract; a token with complex code requires a higher standard of review. Use Slither's `human-summary` printer to identify complex code.
- ❏ **The contract uses `SafeMath`.** Contracts that do not use `SafeMath` require a higher standard of review. Inspect the contract by hand for `SafeMath` usage.
- ❏ **The contract has only a few non-token-related functions.** Non-token-related functions increase the likelihood of an issue in the contract. Use Slither's `contract-summary` printer to broadly review the code used in the contract.

- ❏ **The token has only one address.** Tokens with multiple entry points for balance updates can break internal bookkeeping based on the address (e.g., `balances[token_address][msg.sender]` may not reflect the actual balance).

Owner Privileges

- ❏ **The token is not upgradeable.** Upgradeable contracts may change their rules over time. Use Slither's `human-summary` printer to determine if the contract is upgradeable.
- ❏ **The owner has limited minting capabilities.** Malicious or compromised owners can abuse minting capabilities. Use Slither's `human-summary` printer to review minting capabilities, and consider manually reviewing the code.
- ❏ **The token is not pausable.** Malicious or compromised owners can trap contracts relying on pausable tokens. Identify pausable code by hand.
- ❏ **The owner cannot blacklist the contract.** Malicious or compromised owners can trap contracts relying on tokens with a blacklist. Identify blacklisting features by hand.
- ❏ **The team behind the token is known and can be held responsible for abuse.** Contracts with anonymous development teams or teams that reside in legal shelters require a higher standard of review.

ERC20 Tokens

ERC20 Conformity Checks

Slither includes a utility, `slither-check-erc`, that reviews the conformance of a token to many related ERC standards. Use `slither-check-erc` to review the following:

- ❏ **`Transfer` and `transferFrom` return a boolean.** Several tokens do not return a boolean on these functions. As a result, their calls in the contract might fail.
- ❏ **The `name`, `decimals`, and `symbol` functions are present if used.** These functions are optional in the ERC20 standard and may not be present.
- ❏ **`Decimals` returns a `uint8`.** Several tokens incorrectly return a `uint256`. In such cases, ensure that the value returned is below 255.
- ❏ **The token mitigates the known ERC20 race condition.** The ERC20 standard has a known ERC20 race condition that must be mitigated to prevent attackers from stealing tokens.

Slither includes a utility, `slither-prop`, that generates unit tests and security properties that can discover many common ERC flaws. Use `slither-prop` to review the following:

- ❏ **The contract passes all unit tests and security properties from `slither-prop`.** Run the generated unit tests and then check the properties with Echidna and Manticore.

Risks of ERC20 Extensions

The behavior of certain contracts may differ from the original ERC specification. Conduct a manual review of the following conditions:

- ❏ **The token is not an ERC777 token and has no external function call in `transfer` or `transferFrom`.** External calls in the transfer functions can lead to reentrancies.
- ❏ **`Transfer` and `transferFrom` should not take a fee.** Deflationary tokens can lead to unexpected behavior.
- ❏ **Potential interest earned from the token is taken into account.** Some tokens distribute interest to token holders. This interest may be trapped in the contract if not taken into account.

Token Scarcity

Reviews of token scarcity issues must be executed manually. Check for the following conditions:

- ❏ **The supply is owned by more than a few users.** If a few users own most of the tokens, they can influence operations based on the tokens' repartition.
- ❏ **The total supply is sufficient.** Tokens with a low total supply can be easily manipulated.
- ❏ **The tokens are located in more than a few exchanges.** If all the tokens are in one exchange, a compromise of the exchange could compromise the contract relying on the token.
- ❏ **Users understand the risks associated with a large amount of funds or flash loans.** Contracts relying on the token balance must account for attackers with a large amount of funds or attacks executed through flash loans.
- ❏ **The token does not allow flash minting.** Flash minting can lead to substantial swings in the balance and the total supply, which necessitate strict and comprehensive overflow checks in the operation of the token.

ERC721 Tokens

ERC721 Conformity Checks

The behavior of certain contracts may differ from the original ERC specification. Conduct a manual review of the following conditions:

- ❏ **Transfers of tokens to the `0x0` address revert.** Several tokens allow transfers to `0x0` and consider tokens transferred to that address to have been burned; however, the ERC721 standard requires that such transfers revert.
- ❏ **`safeTransferFrom` functions are implemented with the correct signature.** Several token contracts do not implement these functions. As a result, a transfer of NFTs to one of those contracts can result in a loss of assets
- ❏ **The `name`, `decimals`, and `symbol` functions are present if used.** These functions are optional in the ERC721 standard and may not be present.
- ❏ **If it is used, `decimals` returns a `uint8(0)`.** Other values are invalid.
- ❏ **The `name` and `symbol` functions can return an empty string.** This behavior is allowed by the standard.
- ❏ **The `ownerOf` function reverts if the `tokenId` is invalid or is set to a token that has already been burned.** The function cannot return `0x0`. This behavior is required by the standard, but it is not always properly implemented.
- ❏ **A transfer of an NFT clears its approvals.** This is required by the standard.
- ❏ **The token ID of an NFT cannot be changed during its lifetime.** This is required by the standard.

Common Risks of the ERC721 Standard

To mitigate the risks associated with ERC721 contracts, conduct a manual review of the following conditions:

- ❏ **The `onERC721Received` callback is taken into account.** External calls in the transfer functions can lead to reentrancies, especially when the callback is not explicit (e.g., in `safeMint` calls).
- ❏ **When an NFT is minted, it is safely transferred to a smart contract.** If there is a minting function, it should behave similarly to `safeTransferFrom` and properly handle the minting of new tokens to a smart contract. This will prevent a loss of assets.

- ❑ **The burning of a token clears its approvals.** If there is a burning function, it should clear the token's previous approvals.