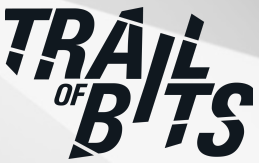

Perpetual Protocol V2

Security Assessment

March 21, 2022

Prepared for:

Perpetual Finance

Prepared by: **Michael Colburn, Paweł Płatek**

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.

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

Table of Contents

Executive Summary

Engagement Overview

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 scope, timeline, test targets, and coverage are provided in the original audit report.

Perpetual Finance contracted Trail of Bits to review the fixes implemented for issues identified in the original report. On March 21, 2022, a team of two consultants conducted a review of the client-provided source code.

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

Overview of Retest Results

Perpetual Finance has sufficiently addressed a few of the issues described in the original audit report.

Project Summary

Contact Information

The following managers were associated with this project:

Dan Guido, Account Manager
dan@trailofbits.com

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

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
March 31, 2022	Delivery of retest report

Project Methodology

Our work in the retesting project included the following:

- A review of the findings in the original audit report
- A manual review of the client-provided source code and fix-related documentation

Project Targets

The engagement involved retesting 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

Summary of Retest Results

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

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

Detailed Retest Results

1. Lack of zero-value checks on functions

Status: **Unresolved**

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)

Fix Analysis

The Perpetual Finance team acknowledged the issue and decided to postpone the fix.

2. Solidity compiler optimizations can be problematic

Status: **Resolved**	
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.

Fix Analysis

The Perpetual Finance team acknowledged the issue.

3. mulDiv reverts instead of returning MIN_INT

Status: **Resolved**	
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)

Fix Analysis

The Perpetual Finance team concluded that the revert is expected.

4. Discrepancies between code and specification

Status: **Resolved**	
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.

Fix Analysis

The Perpetual Finance team concluded that discrepancies between the code and specification are intended because the discrepancies are only implementation details.

5. Missing Chainlink price feed safety checks

Status: **Partially Resolved**	
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.

Fix Analysis

The first and third bullets from the description have been resolved: Perpetual Finance's smart contracts are supposed to use Chainlink's `AggregatorFacade` contract, which includes the relevant safety checks. If Perpetual Finance decides to change the Chainlink contract, then the safety checks will have to be implemented.

For the `latestPrice > 0` check, the Perpetual Finance team decided that returning a zero price is better than pausing the system.

For the `require(latestTimestamp > baseTimestamp)` check, the Perpetual Finance team indicated that the "Chainlink contract returns the same value of `startedAt` and `updatedAt`." However, the `baseTimestamp` is not returned from the Chainlink contract; rather, a block's timestamp minus the TWAP interval is returned. So the team's response does not address the issue that the oracle may return a very outdated price.

The Perpetual Finance team did not address the problem of "not monotonic" `roundIds`.

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

Status: **Partially Resolved**	
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`.

Fix Analysis

According to the Perpetual Finance team's response, the first edge case (in which there is not enough historical data and an entry in the `observations` array is empty) is handled manually: the team will open new markets only after enough historical data is acquired.

For the second edge case (in which the historical data is very old), the Perpetual Finance team responded with the following: "Also, the current implementation seems correct, as this is what the function `_getSurroundingObservations` is expected to do: return the two closest timestamp to our target one!" However, the issue does not concern the correctness of the implementation, but describes a general risk that the price feed could return outdated prices.

7. Ever-increasing priceCumulative variables

Status: **Resolved**	
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`)

Fix Analysis

This issue was fixed in PR#20 with the use of `SafeMath` for relevant arithmetic operations.

8. Lack of rounding in Emergency price feed

Status: **Unresolved**	
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)

Fix Analysis

The Perpetual Finance team acknowledged the issue and decided to postpone the fix.

9. It is possible to pollute the observations array

Status: **Undetermined**	
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)

Fix Analysis

The finding was not considered during the retest because the vulnerability is relevant only for a newer version of the code that was not part of the original audit.

A. Status Categories

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

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

B. Vulnerability Categories

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

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

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.