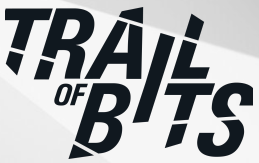

Perpetual Protocol V2

Security Assessment

March 4, 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 such, 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.

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 3 consultants conducted a security review of the client-provided source code, with 6 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	3
Low	0
Informational	5
Undetermined	0

CATEGORY BREAKDOWN

Category	*Count*
Data Validation	5
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

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 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` is the main entrypoint of the protocol for users. It allows users to manage liquidity, open and close long or short positions as well as 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 were 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 the interactions with Uniswap were done properly, that orders were tracked consistently internally, and that fee amounts were 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 was consistent and calculations were sound.

- `Vault`. The Vault contract is the place where user collateral is stored. We reviewed this contract to ensure tokens are handled properly within it and 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 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 make use of safe arithmetic libraries to prevent overflows and ensure safe casting between integer types. Any instances that don't use these libraries either cannot overflow or have comments indicating this is intended behavior.	**Strong**
Auditing	Many functions in the contracts emit events when necessary. We also encourage the development of an incident response plan and implementing on-chain monitoring if this has not already been done.	**Satisfactory**
Authentication / Access Controls	There are appropriate access controls in place for privileged operations, both for operations between contracts as well as contract owner functionality.	**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 result in some interactions at times being difficult to follow as they pass between different contracts.	**Satisfactory**
Cryptography and Key Management	This 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. The contracts are also deployed behind proxies which allows for the implementations to be upgraded in the future. Ownership of the contracts is transferred to a Gnosis multisig at the end of the deployment process.	**Satisfactory**
Documentation	The project has good high-level documentation, a specification where the derivations for formulas used by the system are shown, as well as good coverage of NatSpec and in-line comments.	**Satisfactory**
Front-Running Resistance	We did not identify any front-running issues during this review. The Perpetual Finance team noted other front-running issues they were already aware of.	**Further investigation required**
Low-Level Calls	The contracts do not use assembly or make low-level calls aside from their use of delegatecall proxies.	**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 through 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 the code and specification	Undefined Behavior	Informational
5	Chainlink price feed safety checks are missing	Data Validation	Medium
6	Band price feed may return invalid price 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

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 `setClearingHouse` and `setExchange` functions of the `ClearingHouseCallee` and `OrderBook` contracts, respectively, 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 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 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 may fail 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 Primitive 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 can't return the minimum signed value (-2^255) – it reverts in such cases.

The function takes as arguments signed numbers, internally operates on the unsigned type, and at the end 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 (perp-lushan/contracts/lib/PerpMath.sol#84)

The `neg256` function converts a number to signed type and negates it. The `toInt256` method checks if the number is less or equal to the maximal signed value (2^255 - 1). So if the unsigned result of value 2^255 is passed to the `neg256`, then it 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 won't cause unexpected reverts.

4. Discrepancies between the 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 as:

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

However, in `ClearingHouse.getAccountValue`, the `fundingPayment` amount is subtracted instead.

In the Liquidation section of the specification, the liquidation fee is calculated as:

```
liquidationFee = exchangePositionNotion * 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. Chainlink price feed safety checks are missing

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

Description

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

- `require(updatedAt > 0)` – this checks if the requested round is valid and completed; an example usage of the check can be found in historical-price-feed-data project, and description of the parameter validation in Chainlink's documentation.
- `latestPrice > 0` – it should be expected that price is greater than zero, but currently the code may consume zero prices, as shown in figure 5.1 and 5.2. This check should be added before any `return` calls.

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

Figure 5.1: Returning zero price is possible (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: Returning zero price is possible (perp-oracle/contracts/ChainlinkPriceFeed.sol#95–99)

- `require(answeredInRound == roundID)` – as 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 can be seen in figure 5.1. The code should revert if no fresh price is available.

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

- if they are decreasing - as shown e.g., in the historical-price-feed-data project
- If they are zero if expected - `updatedAt` may equal to zero for missing rounds, and that may indicate that requested `roundId` is invalid. For example, see these checks in the historical-price-feed-data project. Please note that requesting invalid round 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 a 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, consider adding a requirement for a minimum number of calls to the `_getRoundData` method, to make sure 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.

6. Band price feed may return invalid price 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:

If there is not enough historical data and some entry in the `observations` array is empty. As shown in figure 6.1, in the case of not enough data, the code reverts. However, if at the same time some `observations` entry is empty, the code will not revert – as shown in figure 6.2 – but will continue execution taking the branch shown 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: 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: Not enough historical data, but condition from figure 6.1 won't 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. In other words, it is easier than users would expect 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 a single, oldest observation and a 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 in case of old data (perp-oracle/contracts/BandPriceFeed.sol#105–139)

We have `lastestObservation == atOrAfter`, so we can reduce the equation:

```
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: Reducing equations in case of old data.

The `currentTimestamp - latestBandData.lastUpdatedBase` should be small, and the `latestBandData.lastUpdatedBase - lastestObservation.timestamp` can be large. So the decisive influence on the final price will have the oldest observation - the `lastestObservation.price` variable. Moreover, if the last update of the Band price was done in the same block as the call to the `getPrice`, then we have `currentTimestamp - latestBandData.lastUpdatedBase == 0` and so the returned price will be exactly the outdated `lastestObservation.price`.

Exploit Scenario

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

Recommendations

Short term, implement following:

- Always revert in case of "not enough historical data", even if there are not enough observations yet. Change condition in figure 6.3 to handle only case of equality, and revert if `targetTimestamp < beforeOrAt.timestamp`.
- Always revert in case of "the latest data is older than requested", handle only the case when the latest data is equal to the requested.

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, `Observation.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 check for overflow nor a way to reset the variable (or 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 large, incorrect price for some token. The 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 wrong, large prices for the token. An attacker exploits that to drain protocol.

Recommendations

Short term, add a check for an overflow for `priceCumulative` variables. Disable price feed in such a case, 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 doesn't round down negative arithmetic mean tick, 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 computation of arithmetic mean tick (perp-oracle/contracts/EmergencyPriceFeed.sol#65–66)

Computations performed in the `OracleLibrary`'s `consult` method can be seen in figure 8.2.

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

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

Recommendations

Short term, consider rounding down negative arithmetic mean ticks before calling `getSqrtRatioAtTick` method.

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 areas marked with TODOs as well as integrating automated testing into the development workflow.
- Consider integrating Slither into your CI process.
- Regularly review and update the documentation to ensure it matches the current codebase.
- If it is not already in place, consider setting up blockchain monitoring to detect any issues with oracles, potentially malicious liquidations, and similar scenarios.
- Develop an incident response plan to address the above scenarios as well as 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 your issue tracker.

perp-oracle/BandPriceFeed

- There is a typo in the variable `lastestObservation` in the `getPrice` function. It should be updated to the `latestObservation`.
- The code relays on overflows to work correctly. If the solidity compiler will be updated to version >= 0.8.0, the code will break. Example place of the issue is presented in figure C.1.

```
// 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 immutable as a minor gas optimization.

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 candidate 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 would encourage the Perpetual Finance to consider doing so going forward.

VirtualToken

- Addresses not whitelisted cannot send tokens
- `totalySupply` is fixed at `uint256(max)`
- Only the contract owner can add or remove users from the whitelist

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 >= getTotalPositionValue`

InsuranceFund

- The contract's balance does not decrease if no `borrower` has been set

E. Token Integration Checklist

Follow this checklist when interacting with arbitrary tokens. Make sure you understand the risks associated with each item, and justify any exceptions to these rules. An up-to-date version of the checklist can be found in crytic/building-secure-contracts.

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

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

To follow this checklist, you'll want to have this 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 that reside in legal shelters should require a higher standard of review.

ERC-20 tokens

Suggested ERC-20 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 that:

- ❏ **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 in ERC-20 extensions

Contracts might have different behaviors from their original ERC specification, manually review that:

- ❏ **The token is not an ERC777 token and has no external function call in `transfer` and `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 might be trapped in the contract if not taken into account.

Token Scarcity
Reviews for issues of token scarcity require manual review. Check for these conditions:

- ❏ **No user owns most of the supply.** If a few users own most of the tokens, they can influence operations based on the token's 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.

ERC-721 tokens

Suggested ERC-721 conformity checks
Contracts might have different behavior from their original ERC specification. Manually review that:

- ❏ **Transferring tokens to 0x0 reverts.** Several tokens allow transferring to 0x0, counting these as burned, however the ERC-721 standard requires a revert in such cases.
- ❏ **`safeTransferFrom` functions are implemented with the correct signature.** Several tokens do not implement these functions. As a result, transferring NFTs into a contract can result in loss of assets
- ❏ **The `name`, `decimals`, and `symbol` functions are present if used.** These functions are optional in the ERC721 standard and might not be present.
- ❏ **If used, `decimals` should return `uint8(0)`.** Other values are not valid.

- ❏ **The name and symbol functions can return an empty string.** This is allowed by the standard.
- ❏ **`ownerOf` reverts if the `tokenId` is invalid or it was already burned.** This function cannot return 0x0. This is required by the standard, but it is not always properly implemented.
- ❏ **Transferring an NFT clears its approvals.** This is required by the standard.
- ❏ **The token id from an NFT cannot be changed during its lifetime.** This is required by the standard.

Common risks to ERC-721

ERC-721 contracts are exposed to the following risks when implementing:

- ❏ **The `onERC721Received` callback is commonly exploited.** External calls in the transfer functions can lead to reentrancies. This is particularly risky when the callback is not explicit. E.g., when calling `safeMint`.
- ❏ **Minting safely transfers an NFT to a smart contract.** If there is a function to mint, it should properly handle minting new tokens to a smart contract (similarly to `safeTransferFrom`) to avoid loss of assets.
- ❏ **Burning clears approvals.** If there is a function to burn, it should clear previous approvals.