Veridise Auditing Report

Hardening Blockchain Security with Formal Methods

FOR

PERENNIAL

Veridise Inc.

May 20, 2023

► **Prepared For:**

Perennial Labs
https://perennial.finance/

► **Prepared By:**

Bryan Tan
Ajinkya Rajput

► **Contact Us:** contact@veridise.com

► **Version History:**

May 20, 2023 V1

Contents

1 Executive Summary

From Apr. 17, 2023 to May 5, 2023, Perennial Labs engaged Veridise to review the security of their Perennial protocol. The review covered the smart contract implementations of three components of their protocol: the core protocol, the oracle, and the balanced vault. Veridise conducted the assessment over 6 person-weeks, with 2 engineers reviewing code over 3 weeks on commit `d579241`. The auditing strategy involved a tool-assisted analysis of the source code performed by Veridise engineers as well as extensive manual auditing.

Code assessment. The Perennial protocol is an automated market maker (AMM) protocol that allows arbitrary users to create customizable derivatives markets called "products" (these users are called *product coordinators*). A product allows liquidity providers (called *makers*) and leveraged traders (called *takers*) to provide assets (specified by the protocol) to trade derivatives positions against each other.

The Veridise auditors reviewed three parts of this protocol: the core protocol, the oracle module, and the balanced vaults. In addition to the product functionality, the core protocol also provides mechanisms for depositing and withdrawing collateral; for configuring the protocol and products; a "funding" mechanism that transfers a fee from takers to makers; as well as an incentive program feature that allows a product coordinator to grant tokens to participants in a product. In the Perennial protocol, orders are not executed immediately; instead, orders are settled only on oracle price feed updates. This price feed data is provided by the oracle module, which reads Chainlink price feed information and transforms it to a format supported by Perennial. Finally, the protocol also has a "balanced vaults" module that allows makers to pool funds into a permissionless "vault" contract, which then algorithmically invests into markets (each consisting of a pair of long/short products on the same derivative).

The Perennial developers provided the source code of the Perennial contracts for review. The source code contained a test suite, which the Veridise auditors noted consist of a combination of unit and integration tests. Several files in the source code also indicate that the developers use linting and static analysis tools such as ESLint and Slither.

To facilitate the Veridise auditors' understanding of the code, the Perennial developers shared their documentation website* with the auditors. The source code also contained some documentation in the form of READMEs and documentation comments on functions and storage variables.

Summary of issues detected. The audit uncovered 33 issues, 2 of which are assessed to be of high or critical severity by the Veridise auditors. Specifically, a malicious product coordinator can change their product's oracle at any time to their own benefit (V-VUL-PER-001), and the balanced vault claim pro-rating for insolvent vaults could result in unrecoverable losses for users (V-VUL-PER-001). The Veridise auditors also identified 6 medium-severity issues, including a potential reentrancy attack vector that can be exploited by a malicious product coordinator

* `https://docs.perennial.finance`

(V-VUL-PER-003) and a bug in the balanced vault that may lead to unintended reverts (V-VUL-PER-007), as well as 7 low-severity issues and 15 warnings. The Perennial developers resolved all of the issues.

Disclaimer. We hope that this report is informative but provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the system is secure in all dimensions. In no event shall Veridise or any of its employees be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the results reported here.

2 Project Dashboard

Table 2.1: Application Summary.

Name	Version	Type	Platform
Perennial	`d579241`	Solidity	Ethereum

Table 2.2: Engagement Summary.

Dates	Method	Consultants Engaged	Level of Effort
Apr. 17 - May 5, 2023	Manual & Tools	2	6 person-weeks

Table 2.3: Vulnerability Summary.

Name	Number	Resolved
Critical-Severity Issues	0	0
High-Severity Issues	2	2
Medium-Severity Issues	6	6
Low-Severity Issues	7	7
Warning-Severity Issues	15	15
Informational-Severity Issues	3	3
TOTAL	33	33

Table 2.4: Category Breakdown.

Name	Number
Data Validation	8
Maintainability	8
Logic Error	7
Gas Optimization	3
Theft	2
Denial of Service	2
Locked Funds	1
Reentrancy	1
Hash Collision	1

3 Audit Goals and Scope

3.1 Audit Goals

The engagement was scoped to provide a security assessment of Perennial's smart contracts. In our audit, we sought to answer the following questions:

- Are the access controls correctly implemented in the core protocol?
- Are malicious product owners able to steal funds from users?
- Can malicious product owners abuse configuration values in a way that harms users?
- Are there any discrepancies in the bookkeeping of the core protocol?
- Does the oracle module correctly address edge cases when interacting with Chainlink?
- Do balanced vaults correctly interact with the core protocol?
- Are there any discrepancies in the bookkeeping of the balanced vaults?

3.2 Audit Methodology & Scope

Audit Methodology. To address the questions above, our audit involved a combination of human experts and automated program analysis & testing tools. In particular, we conducted our audit with the aid of the following techniques:

- *Static analysis.* To identify potential common vulnerabilities, we leveraged our custom smart contract analysis tool Vanguard, as well as the open-source tool Slither. These tools are designed to find instances of common smart contract vulnerabilities, such as reentrancy and uninitialized variables.
- *Fuzzing/Property-based Testing.* We also leverage fuzz testing to determine if the protocol may deviate from the expected behavior. To do this, we formalize the desired behavior of the protocol as [V] specifications and then use our fuzzing framework OrCa to determine if a violation of the specification can be found.

Scope. The scope of this audit is limited to the Solidity source files of the `packages` folder of the source code provided by the Perennial developers, which contains the smart contract implementation of the Perennial. Specifically, the auditors considered the files in the following folders:

- `packages/perennial/contracts`
- `packages/perennial-oracle/contracts`
- `packages/perennial-vaults/contracts`

During the audit, the Veridise auditors observed that the code in the scope of the audit references or uses code written by the Perennial developers or by third-parties that is *not* in the scope of the audit. The auditors referred to these excluded files but assumed that they have been implemented correctly.

Methodology. Veridise auditors reviewed the reports of previous audits for Perennial, inspected the provided tests, and read the Perennial documentation. They then began a manual audit of the code assisted by both static analyzers and automated testing. During the audit, the Veridise auditors regularly met with the Perennial Labs developers to ask questions about the code.

3.3 Classification of Vulnerabilities

When Veridise auditors discover a possible security vulnerability, they must estimate its severity by weighing its potential impact against the likelihood that a problem will arise. Table 3.1 shows how our auditors weigh this information to estimate the severity of a given issue. We judge the likelihood of a vulnerability as follows in Table 3.2. In addition, we judge the impact of a vulnerability as follows in Table 3.3.

Table 3.1: Severity Breakdown.

	Somewhat Bad	Bad	Very Bad	Protocol Breaking
Not Likely	Info	Warning	Low	Medium
Likely	Warning	Low	Medium	High
Very Likely	Low	Medium	High	Critical

Table 3.2: Likelihood Breakdown

Not Likely	A small set of users must make a specific mistake
Likely	Requires a complex series of steps by almost any user(s) - OR - Requires a small set of users to perform an action
Very Likely	Can be easily performed by almost anyone

Table 3.3: Impact Breakdown

Somewhat Bad	Inconveniences a small number of users and can be fixed by the user
Bad	Affects a large number of people and can be fixed by the user - OR - Affects a very small number of people and requires aid to fix
Very Bad	Affects a large number of people and requires aid to fix - OR - Disrupts the intended behavior of the protocol for a small group of users through no fault of their own
Protocol Breaking	Disrupts the intended behavior of the protocol for a large group of users through no fault of their own

4 Vulnerability Report

In this section, we describe the vulnerabilities found during our audit. For each issue found, we log the type of the issue, its severity, location in the code base, and its current status (i.e., acknowleged, fixed, etc.). Table 4.1 summarizes the issues discovered:

Table 4.1: Summary of Discovered Vulnerabilities.

ID	Description	Severity	Status
V-VUL-PER-001	Malicious product owner can arbitrarily change . . .	High	Acknowledged
V-VUL-PER-002	Users may lose funds if claim() pro-rating trig. . .	High	Acknowledged
V-VUL-PER-003	Reentrancy attack vector through incentive prog. . .	Medium	Fixed
V-VUL-PER-004	Malicious product owner can force liquidation b. .	Medium	Acknowledged
V-VUL-PER-005	Program completes with last settled version dur. . .	Medium	Intended Behavior
V-VUL-PER-006	Potential subtraction overflow in calculating p. . .	Medium	Intended Behavior
V-VUL-PER-007	Vault collateral rebalance can revert if withdr. . .	Medium	Fixed
V-VUL-PER-008	Unenforced assumption that market product pair .	Medium	Fixed
V-VUL-PER-009	Accounts can be settled when protocol is paused	Low	Fixed
V-VUL-PER-010	Product owner can DoS new or inactive users by . .	Low	Won't Fix
V-VUL-PER-011	Unspecified Solidity compiler behavior in Multi. . .	Low	Fixed
V-VUL-PER-012	Potential selector collision in MultiInvokerRol. . .	Low	Fixed
V-VUL-PER-013	MultiInvoker withdraw & unwrap reverts when w	Low	Fixed
V-VUL-PER-014	Missing parameter validation in BalancedVault c. .	Low	Fixed
V-VUL-PER-015	BalancedVault does not initialize oracle versio. . .	Low	Fixed
V-VUL-PER-016	Inconsistent Solidity compiler versions	Warning	Fixed
V-VUL-PER-017	Risk of encountering known Solidity compiler bug	Warning	Fixed
V-VUL-PER-018	Invalid product parameter defaults to protocol . . .	Warning	Fixed
V-VUL-PER-019	Dangerous memory aliasing in deductFee	Warning	Fixed
V-VUL-PER-020	Error-prone code duplication in MultiInvokerRollu	Warning	Fixed
V-VUL-PER-021	Significant gas costs due to MultiInvokerRollup. . .	Warning	Won't Fix
V-VUL-PER-022	Double check non-negative price assumption	Warning	Fixed
V-VUL-PER-023	No validation in Incentivizer.owner	Warning	Fixed
V-VUL-PER-024	BalancedVault allows deposit to address 0	Warning	Won't Fix
V-VUL-PER-025	No checks against oracle updated timestamp of zer	Warning	Fixed
V-VUL-PER-026	No check that starting round ID is valid	Warning	Acknowledged
V-VUL-PER-027	Error-prone initialization in constructor of up. . .	Warning	Fixed
V-VUL-PER-028	ReservoirFeedOracle does not validate starting . . .	Warning	Fixed
V-VUL-PER-029	getStartingRoundId assumes phase is invalid if . . .	Warning	Acknowledged
V-VUL-PER-030	Shares are accumulated when product is closed	Warning	Intended Behavior
V-VUL-PER-031	Potential extra memory copy in _claimProduct	Info	Fixed
V-VUL-PER-032	Unnecessary memory allocations in MultiInvokerR	Info	Fixed
V-VUL-PER-033	Settlement assumes contiguity of oracle versions	Info	Intended Behavior

4.1 Detailed Description of Issues

4.1.1 V-VUL-PER-001: Malicious product owner can arbitrarily change oracle contract

Severity	High	**Commit**	d579241
Type	Theft	**Status**	Acknowledged
File(s)	`Product.sol`,`UPayoffProvider.sol`		
Location(s)	updateOracle(), _updateOracle()		

The Perennial protocol allows any user to become a "coordinator" that can launch their own Product contract. The `Product.updateOracle()` function allows the coordinator of a product to dynamically change the Perennial `IOracleProvider` contract that is used to obtain price information. Because a coordinator can be an arbitrary untrusted user, a malicious product coordinator can use this functionality to steal funds from users of their Product contract.

```
function updateOracle(IOracleProvider newOracle) external onlyProductOwner {
    _updateOracle(address(newOracle), latestVersion());
}
```

Snippet 4.1: Definition of `Product.updateOracle()`

```
function _updateOracle(address newOracle, uint256 oracleVersion) internal {
    if (!Address.isContract(newOracle)) revert PayoffProviderInvalidOracle();
    _oracle.store(newOracle);

    emit OracleUpdated(newOracle, oracleVersion);
}
```

Snippet 4.2: Definition of `UPayoffProvider._updateOracle()`

Theoretical Attack Scenario

1. An attacker becomes a coordinator and deploys a Product contract. They run the Product legitimately for a period of time in order to earn users' trust. Alternatively, an attacker could steal the private keys of the account that is marked as the Product's coordinator.
2. The attacker opens a large position using a different account. In particular, to avoid the "socialization factor" limitations on makers, the attacker may be inclined to open a taker position.
3. Right before settlement occurs, the attacker then updates the oracle with a malicious one deployed by the attacker. The malicious oracle uses a price that is advantageous to the attacker's position. The attacker can then close their position in order to realize a profit.

Impact The attacker can steal funds of users in the product by manipulating prices in a way that allows the attacker's positions to always profit. This will harm users that take the side opposite the attacker's position. Furthermore, makers are more likely to be harmed than takers,

as the attack is less effective when the attacker takes a maker position (rather than a taker position).

Recommendation There are several possible mitigations, but each with different trade-offs.

- To avoid such an attack altogether, the developers could remove the functionality that allows the oracle provider address to be updated. However, this may lead to issues if the oracle contract is compromised or suffers from a bug.
- To allow upgradability but mitigate the damage of such an attack, the `Controller` contract can be extended so that oracle provider contracts must be deployed with a BeaconProxy, similar to `Controller.createProduct()`. The `Product` contract can then enforce that an oracle provider address can only be updated to those created by the `Controller`. Note, however, that this 1) adds additional complexity to the protocol; 2) does not solve the problem of a malicious product owner being able to swap out oracle providers in a malicious way; and 3) assumes that each whitelisted oracle provider has sufficient parameter validation so that it cannot be configured in a malicious way.

Developer Response The developers responded:

> We will not fix this in this version of the protocol. The dApp and users will need to make informed decisions about using products with unknown and/or untrusted product coordinators.

When asked why the oracle contract can be changed, the developers said that they need the oracle update functionality in case an oracle has a bug.

4.1.2 V-VUL-PER-002: Users may lose funds if claim() pro-rating triggers

Severity	High	**Commit**	d579241
Type	Locked Funds	**Status**	Acknowledged
File(s)	`BalancedVault.sol`		
Location(s)	claim()		

A user can convert their shares in a `BalancedVault` into collateral by calling the `redeem()` method, which updates the bookkeeping in the `_unclaimed` mapping for that user and the `_totalUnclaimed` across all users. After the next settlement, users get their collateral using the `claim()` method. The method first retrieves the total unclaimed amount of the vault in `unclaimedTotal` and the

```
function claim(address account) external {
  (EpochContext memory context, ) = _settle(account);

  UFixed18 unclaimedAmount = _unclaimed[account];
  UFixed18 unclaimedTotal = _totalUnclaimed;
  _unclaimed[account] = UFixed18Lib.ZERO;
  _totalUnclaimed = unclaimedTotal.sub(unclaimedAmount);
  emit Claim(msg.sender, account, unclaimedAmount);

  // pro-rate if vault has less collateral than unclaimed
  UFixed18 claimAmount = unclaimedAmount;
  UFixed18 totalCollateral = _assets();
  if (totalCollateral.lt(unclaimedTotal)) claimAmount = claimAmount.muldiv(
    totalCollateral, unclaimedTotal);

  _rebalance(context, claimAmount);

  asset.push(account, claimAmount);
}
```

Snippet 4.3: Definition of `claim()`

unclaimed amount of a particular user in `unclaimedAmount`. Then it zeroes out the unclaimed amount of the user and deducts the amount from the `_totalUnclaimed`.

The actual `claimAmount` to be transferred to the user depends on whether the vault has enough collateral to actually pay out the `unclaimedAmount`. If there are enough `_assets()` (consisting of the vault's instantaneous token balance plus any collateral that can be withdrawn from the vault) to cover the withdrawal, then the actual `claimAmount` will be the `unclaimedAmount`.

Otherwise, the withdrawn amount will be *pro-rated* so that the actual `claimAmount` is scaled by the proportion of funds that can actually be distributed to all users, `totalCollateral / unclaimedTotal`. In this case, the user will only receive `claimAmount * totalCollateral / unclaimedTotal`, but their entire `_unclaimed` entry will have been zeroed out, causing them to lose the remaining funds.

Additionally, the `Claim` event only reports the `unclaimedAmount`, but not the actual `claimAmount` transferred to the user.

Impact If the `totalCollateral` is less than `unclaimedTotal`, then the user will only receive the pro-rated amount of collateral with no possibility to recover the remaining amount of collateral, even if someone "rescues" the vault by adding enough collateral for all users to be made whole.

Recommendation To allow users to recover all of their funds in the event of a shortfall, the developers may want to consider other strategies, which may include:

- A strategy like first-come-first-serve (FCFS) that does not result in loss of funds (assuming the vault is rescued); however, these also have other tradeoffs in terms of fairness.
- Introduce additional bookkeeping that tracks the amount that is "lost" when funds are pro-rated, so that they can later be claimed if the vault become solvent again. However, this may add complexity (and room for error) to the claim process.

Developer Response The developers agreed that the loss of funds could occur, but they will "opt to keep the pro-rata logic as we believe it is the most fair versus FCFS."

4.1.3 V-VUL-PER-003: Reentrancy attack vector through incentive program rewards

Severity	Medium	Commit	d579241
Type	Reentrancy	Status	Fixed
File(s)	`Incentivizer.sol`,`Product.sol`		
Location(s)	Product._settle(), Incentivizer._handleSyncResult()		

During global settlement of a Product in `_settle()`, any rewards program that is running in that product will be updated by calling `Incentivizer.sync()`. After updating the reward program bookkeeping, the `sync()` method will invoke `_handleSyncResult()` on each reward program. If a reward program has completed, then any remaining tokens used for the reward program will be transferred to the product's treasury. It may be possible for an malicious product coordinator

```
function _handleSyncResult(IProduct product, ProductManagerLib.SyncResult memory
    syncResult) private {
    uint256 programId = syncResult.programId;
    if (!syncResult.refundAmount.isZero())
        _products[product].token(programId).push(treasury(product, programId),
    syncResult.refundAmount);
```

Snippet 4.4: Location in `Incentivizer._handleSyncResult()` that transfers tokens to the product's treasury.

to abuse the refund functionality if:

- The product coordinator sets the reward token to an ERC20 subtype that supports callbacks, such as an ERC777 token, or an ERC20 that wraps a callback-supporting token such as an ERC1155.
- The product treasury is a malicious smart contract deployed by the coordinator.

For example, the treasury can be used to launch reentrancy attacks or perform denial-of-service attacks against legitimate users of the protocol.

Example Attack Scenario The auditors have managed to construct a concrete example of this attack scenario using the Hardhat test environment provided by the developers. The attack is summarized below:

1. The product coordinator creates a reward program.
2. The reward program starts.
3. The product coordinator becomes malicious (e.g., an attacker gains unauthorized access to the product coordinator's account) and updates the product treasury to point to an attacker contract.
4. Suppose enough time has passed so that the reward program is now complete, but the state of the reward program in the protocol has not been updated yet. Further assume that there remain some nonzero amounts of the reward token, and that there is a new oracle version.
5. The product coordinator calls `Controller.updateCoordinatorPendingOwner()` to initiate coordinator ownership transfer to the attacker contract.
6. Anyone calls any of the user-flows on `Product` that forces settlement first (i.e., a method that invokes `_settle()`). Note that this will activate the reentrancy guard on `Product`.

7. `Product._settle()` calls `Incentivizer.sync()`
8. Because the reward program is now complete, a nonzero refund amount of the reward token is sent to the product coordinator's treasury (i.e., the attacker contract).
9. The ERC20 token invokes a callback on the attacker contract.
10. Because control flow has returned to the attacker, the attacker can now perform malicious actions.

Impact We note that this attack is fairly difficult to perform because it requires *all* of the following conditions to be satisfied:

- The attacker has access to the product coordinator's account.
- The reward ERC20 token is able to invoke callbacks on the recipient of a transfer. Note that the product coordinator is free to choose what reward token they want to use when they create a reward program, including an ERC20-compliant smart contract controlled by the attacker.
- The attack can only be triggered when a reward program is "completed" but is not marked as completed by the incentivizer.
- The reward program must have a nonzero amount of tokens to refund to the product coordinator's treasury; however, this is likely to be the case due to how reward program start and complete times are quantized to the oracle timestamps.

Using this attack vector, the attacker is able to perform actions such as:

- The attacker contract can revert on all calls. This allows the product coordinator to perform a denial-of-service (DoS) attack and prevent any settlements (and therefore user flows) from occurring. Consequently, users will be unable to close or liquidate positions in the product, thereby locking user funds.
- The attacker contract can call `acceptCoordinatorOwner()` to become the new product coordinator. This allows it to call `Product.updateClosed()`, which has no reentrancy guard but calls `_settle()`. That is, it is possible to reenter into the `_settle()` method while a `_settle()` is already being invoked, which can corrupt the protocol's bookkeeping and/or cause double-counting issues.
- The attacker contract can change the oracle before settlement proceeds to manipulate prices (see V-VUL-PER-001).

Recommendation To prevent this issue and reduce the attack surface of the protocol, we recommend that the developers make the following changes:

- Because the reward token can be arbitrarily chosen, it may be safer to use a "pull" model (rather than the existing "push" model) to distribute reward token refunds to the product coordinator. The refund amount can be added to a mapping (for example, by reusing the existing claims mechanism), and the product coordinator can retrieve their refund in a separate transaction.
- Mark `Product.updateClosed()` and `Product.updateOracle()` as `nonReentrant` to reduce the attack surface.


```
contract AttackerTreasury is RewardRecipient {
    event Received(address token, uint256 amount);
    event Attack();

    IProduct reentryTarget;

        // The attacker will call ‘updateCoordinatorPendingOwner‘ on this contract
    // and then call this method to execute the reentrancy attack.
    function executeAttack(IController controller, IProduct product, uint256
    coordinatorId, uint256 programId) external {
        reentryTarget = product;
        controller.acceptCoordinatorOwner(coordinatorId);
        controller.updateCoordinatorPendingOwner(coordinatorId, msg.sender);
        product.settle();
    }

    // This is an example of a callback that may be executed when the reward
    // tokens are refunded to the product’s treasury (this contract) and the
    // reward token is an ERC20 token that invokes a callback on the recipient.
    function onRewardReceived(uint256 amount) external {
        emit Received(msg.sender, amount);
        if (address(reentryTarget) != address(0)) {
            emit Attack();
            // The updateClosed method will trigger settlement again.
            // It is not protected by a reentrancy guard.
            reentryTarget.updateClosed(true);
        }
    }
}
```

Snippet 4.5: Example of an attacker contract that can reenter into `_settle()`.

4.1.4 V-VUL-PER-004: Malicious product owner can force liquidation by updating maintenance value

Severity	Medium	**Commit**	d579241
Type	Theft	**Status**	Acknowledged
File(s)	`UParamProvider.sol`		
Location(s)	updateMaintenance()		

The coordinator of a product is allowed to update the maintenance value of a Product using the `UParamProvider.updateMaintenance()` method. This maintenance value controls the minimum amount of collateral a user must have (as a multiple of their position) in order to avoid liquidation.

Due to the way `updateMaintenance()` works, a compromised or malicious product coordinator can abuse `updateMaintenance()` in order to harm users of the product:

1. The product coordinator can set the maintenance value to any number, so that they can force liquidations.
2. The maintenance value update takes place immediately, so users have absolutely no warning of when the product coordinator can perform this update.

Impact The above factors allow a compromised or malicious product coordinator to force users to be liquidated in a way that cannot be prevented with on-chain monitoring. For example, in a single transaction, the coordinator can:

1. Update the maintenance value to a large value.
2. Liquidate several makers or takers that have large positions. This will allow the coordinator to gain a portion of their collateral.
3. Set the maintenance value back to the old value.

Recommendation The protocol can adopt several mitigations, including but not limited to:

- Only updating the maintenance value after the next settlement. This may allows on-chain monitoring to detect malicious product coordinators. Users will be able to close their positions rather than be subject to a forced liquidation.
- Allowing the protocol owner to impose a cap on maintenance values.

Developer Response The developers responded with the following:

> We will not fix this in this version of the protocol. The dApp and users will need to make informed decisions about using products with unknown and/or untrusted product coordinators.

The developers also noted that they are planning to introduce countermeasures in the future, such as not allowing product coordinators to increase the maintenance values.

4.1.5 V-VUL-PER-005: Program completes with last settled version during settlement

Severity	Medium	Commit	d579241
Type	Logic Error	Status	Intended Behavior
File(s)	Program.sol		
Location(s)	complete()		

If a reward program is completed during an oracle version update, there will eventually be to a call to `ProgramLib.complete()`, which calculates the pro-rated amount of undistributed rewards to refund to the product's treasury and returns the oracle version at which the program was completed. Specifically, the `versionComplete` is obtained by taking the max of the (previously recorded) oracle version that the program was started at and the last settled oracle version on the product. However, during settlement, a program is completed only when the `Product._settle()`

```
function complete(
    Program storage self,
    IProduct product,
    ProgramInfo memory programInfo
) internal returns (uint256 versionComplete, UFixed18 refundAmount) {
    uint256 versionStarted = self.versionStarted;
    versionComplete = Math.max(versionStarted, product.latestVersion());
    self.versionComplete = versionComplete;

    IOracleProvider.OracleVersion memory fromOracleVersion = product.atVersion(
    versionStarted);
    IOracleProvider.OracleVersion memory toOracleVersion = product.atVersion(
    versionComplete);

    uint256 inactiveDuration = programInfo.duration - (toOracleVersion.timestamp -
    fromOracleVersion.timestamp);
    refundAmount = programInfo.amount.sum().muldiv(inactiveDuration, programInfo.
    duration);
    self.available = self.available.sub(refundAmount);
}
```

Snippet 4.6: Definition of `ProgramLib.complete()`

method invokes `Incentivizer.sync()` to update the state of each reward program. This means that `versionComplete` will be set to `product.latestVersion()` (because the `versionStarted` is at most the `product.latestVersion()`), which is inconsistent with the idea that the version that the program is actually "complete" at is oracle version *after* the `product.latestVersion()`.

Impact This issue has two effects. First, the computed `versionComplete` will be incorrect, which can lead to business logic errors in other locations in the protocol. Second, the refund amount will be *larger* than it is supposed to be, because there will be a smaller difference between the `toOracleVersion.timestamp` and the `fromOracleVersion.timestamp`, thereby increasing the `inactiveDuration`. This can result in users being unable to claim their reward tokens because the refunded amount is too large, or in the worst case, a denial-of-service issue for all users of the product due to `self.available.sub(refundAmount)` reverting.

Recommendation One potential way to fix this is to make `versionComplete` a parameter of the `complete()` function, so that the callers can pass in the intended complete version.

Developer Response The developers noted that rewards can only be distributed once settlement has completed. A program is completed if its end time is after the last settled version's update timestamp. Because the rewards are based on duration, the completed oracle version has to be set to the last settled version to avoid counting the time past the end of the reward program, otherwise excessive rewards would be granted to users.

4.1.6 V-VUL-PER-006: Potential subtraction overflow in calculating program's inactive duration

Severity	Medium	Commit	d579241
Type	Denial of Service	Status	Intended Behavior
File(s)	`Program.sol`		
Location(s)	complete()		

The `ProgramLib.complete()` function is used to update the bookkeeping for a reward program when it is completed. If the product coordinator chooses to end the program early, a prorated amount of the unrewarded tokens will be refunded to the product's treasury, based on the time that the program had remaining before it was completed (`inactiveDuration`). The `inactiveDuration` is calculated by subtracting the duration of the program from the elapsed time between the start and completed oracle versions. However, if the elapsed time is larger than the duration of the program, then there will be a revert due to subtraction overflow. Specifically,

```
function complete(
    Program storage self,
    IProduct product,
    ProgramInfo memory programInfo
) internal returns (uint256 versionComplete, UFixed18 refundAmount) {
    uint256 versionStarted = self.versionStarted;
    versionComplete = Math.max(versionStarted, product.latestVersion());
    self.versionComplete = versionComplete;

    IOracleProvider.OracleVersion memory fromOracleVersion = product.atVersion(
    versionStarted);
    IOracleProvider.OracleVersion memory toOracleVersion = product.atVersion(
    versionComplete);

    uint256 inactiveDuration = programInfo.duration - (toOracleVersion.timestamp -
    fromOracleVersion.timestamp);
    refundAmount = programInfo.amount.sum().muldiv(inactiveDuration, programInfo.
    duration);
    self.available = self.available.sub(refundAmount);
}
```

Snippet 4.7: Definition of `complete()`

we note that this overflow issue can occur if:

- a reward program is completed before the next oracle version; and
- the elapsed time between the completed oracle version and the started oracle version is larger than the duration of the reward program

These conditions may be satisfied if the reward program is short and the oracle has availability problems. We note, however, that there is a minimum duration on a reward program, which helps reduce the likelihood of this issue.

Impact Because the `Incentivizer.sync()` method is called as part of the settlement flywheel, this overflow may cause settlements on the affected `Product` to revert and therefore a denial-of-service issue on all major user flows (open/close make/take, deposit, withdraw, liquidate).

Note that a malicious product owner can intentionally trigger such a situation by creating a long-running reward program.

Recommendation To avoid subtraction overflow, the `inactiveDuration` can be computed as follows:

```
uint256 elapsed = toOracleVersion.timestamp - fromOracleVersion.timestamp;
uint256 inactiveDuration = programInfo.duration > elapsed ? programInfo.duration -
    elapsed : 0;
```

Developer Response The developers noted that the intended behavior of the reward program oracle versions is that the start oracle version will be the first oracle version that occurs after the start time, and the complete oracle version will be the last oracle version settled before the end time. Thus, the time elapsed must be smaller than the duration, and the subtraction cannot overflow.

4.1.7 V-VUL-PER-007: Vault collateral rebalance can revert if withdrawing

Severity	Medium	**Commit**	d579241
Type	Logic Error	**Status**	Fixed
File(s)	`BalancedVault.sol`		
Location(s)	_rebalanceCollateral(), _updateCollateral()		

The `BalancedVault` contract uses the `_rebalance()` method to adjust the asset allocation in each market. As part of this adjustment, the `_rebalanceCollateral()` method will use deposits and withdrawals to ensure that the amount deposited into a market reach a target amount (evenly split between long and short). Specifically, if the target amount of a long/short product is below the current collateral value, the `_updateCollateral()` method will withdraw collateral from that product.

The adjustment amount is purely based on the collateral value of the given account and does not take the maintenance/liquidation threshold into consideration. Thus, the call to `collateral.withdrawTo()` in `_updateCollateral()` may revert if the target amount of collateral is below the required maintenance amount. This is unintended behavior, as a documentation comment on the `_rebalance()` method states that: "Rebalance is executed on best-effort, any failing legs of the strategy will not cause a revert".

Note that the `_rebalance()` will also open/close positions, but this only occurs after the call to `_rebalanceCollateral()`.

Impact Rebalancing may revert unexpectedly on vault settlements as well as the three user flows in the vault (deposit, claim, and redeem), since all of these call `_rebalance()` at some point. This risk is higher if the collateral value of a position has increased since the last rebalance.

Recommendation The developers should change `_updateCollateral()` to account for the maintenance/liquidation threshold.

```
function _rebalanceCollateral(UFixed18 claimAmount) private {
    // Compute target collateral
    UFixed18 targetCollateral = _assets().sub(claimAmount).div(TWO);
    if (targetCollateral.muldiv(minWeight, totalWeight).lt(controller.minCollateral()
    ))
        targetCollateral = UFixed18Lib.ZERO;

    // Remove collateral from markets above target
    for (uint256 marketId; marketId < totalMarkets; marketId++) {
        UFixed18 marketCollateral = targetCollateral.muldiv(markets(marketId).weight,
     totalWeight);

        if (collateral.collateral(address(this), markets(marketId).long).gt(
    marketCollateral))
            _updateCollateral(markets(marketId).long, marketCollateral);
        if (collateral.collateral(address(this), markets(marketId).short).gt(
    marketCollateral))
            _updateCollateral(markets(marketId).short, marketCollateral);
    }

    // Deposit collateral to markets below target
    for (uint256 marketId; marketId < totalMarkets; marketId++) {
        UFixed18 marketCollateral = targetCollateral.muldiv(markets(marketId).weight,
     totalWeight);

        if (collateral.collateral(address(this), markets(marketId).long).lt(
    marketCollateral))
            _updateCollateral(markets(marketId).long, marketCollateral);
        if (collateral.collateral(address(this), markets(marketId).short).lt(
    marketCollateral))
            _updateCollateral(markets(marketId).short, marketCollateral);
    }
}
```

Snippet 4.8: Implementation of `_rebalanceCollateral()`

```
function _updateCollateral(IProduct product, UFixed18 targetCollateral) private {
    UFixed18 currentCollateral = collateral.collateral(address(this), product);

    if (currentCollateral.gt(targetCollateral))
        collateral.withdrawTo(address(this), product, currentCollateral.sub(
    targetCollateral));
    if (currentCollateral.lt(targetCollateral))
        collateral.depositTo(address(this), product, targetCollateral.sub(
    currentCollateral));
}
```

Snippet 4.9: Implementation of `_updateCollateral()`

4.1.8 V-VUL-PER-008: Unenforced assumption that market product pair uses same oracle, payoff function

Severity	Medium	Commit	d579241
Type	Data Validation	Status	Fixed
File(s)	`BalancedVault.sol`		
Location(s)	See description		

The `BalancedVault` defines a "market" as a pair of products, one considered to be a long position and one considered to a short position. According to the Perennial documentation, one functionality of these product pairs is to provide "Partial delta hedging (i.e. exposure on one side of the market is partially netted out by exposure on the other side)".

One assumption about the product pairs that is not apparent in either the documentation or the code is that for a given long/short product pair, both of the products must use the same oracle and payoff function. However, there is no code in the constructor or initialization logic that enforces this assumption.

Impact Because this assumption is both implicit and unenforced, it is easy for deployers of the vault to supply a product pair that that uses different oracles or pricing functions. This may lead to business logic errors such as the following:

1. Any code that assumes that the oracle version of the short product moves with the oracle version of the long product will be inconsistent. During settlement and initialization, the oracle version of an epoch is set to the oracle version of the **long** product. However,

```
1 _marketAccounts[marketId].versionOf[context.epoch] = markets(marketId).long.
      latestVersion();
```

Snippet 4.10: Relevant lines in `_settle()`

   the `currentEpochComplete()` method used to determine whether the current epoch is "complete" only checks against the `_marketAccounts[marketId].versionOf`. Specifically, it considers an epoch to complete if and only if for all markets, the minimum of the respective oracle versions of the long and short ***not*** equal to the epoch's oracle version. If the long and short are using different oracles, this may incorrectly return true when the long's oracle version is past the latest's, but the latest's is past the short's. Such a situation may create epochs when no settlements have occurred.
2. A problem similar to above can be observed in `currentEpochStale()`, except that if the short's oracle version is significantly larger than the long's, then the epoch will almost always be considered "stale". Such a situation will lead to `deposit()` and `redeem()` always being delayed by one epoch.
3. When calculating the target amount of positions when rebalancing, the target position amount is calculated in terms of the long product's positions, but is also used to rebalance the positions in the ***short*** product. The target number of short positions may therefore be different than expected.

```
function currentEpochComplete() public view returns (bool) {
    for (uint256 marketId; marketId < totalMarkets; marketId++) {
        if (
            Math.min(markets(marketId).long.latestVersion(), markets(marketId).short.
    latestVersion()) ==
            _versionAtEpoch(marketId, _latestEpoch)
        ) return false;
    }
    return true;
}
```

Snippet 4.11: Implementation of `currentEpochComplete()`

```
UFixed18 currentPrice = markets(marketId).long.atVersion(version).price.abs();
UFixed18 targetPosition = marketCollateral.mul(targetLeverage).div(currentPrice);

_updateMakerPosition(markets(marketId).long, targetPosition);
_updateMakerPosition(markets(marketId).short, targetPosition);
```

Snippet 4.12: Relevant lines in `_rebalancePosition()`

Recommendation The developers should clearly state this assumption in their developer documentation, and they may want to think of a way to enforce this assumption. The current design makes it difficult to check whether a long/short pair have the same oracle and pricing function due to V-VUL-PER-001.

Developer Response The developers added documentation comments to `BalancedVaultDefinition` that states this assumption, and they inserted some additional validation into the constructor. However, the oracle can still differ if a long/short's product owner exploits V-VUL-PER-001.

4.1.9 V-VUL-PER-009: Accounts can be settled when protocol is paused

Severity	Low	**Commit**	d579241
Type	Logic Error	**Status**	Fixed
File(s)	`Collateral.sol`		
Location(s)	settleAccount(), settleProduct()		

The `Collateral.settleProduct()` method is invoked by a `Product` contract to update the global bookkeeping of the protocol. However, it can still be called even when the protocol's pausing mechanism is engaged. Specifically, `Product.updateClosed()` will invoke `_settle()`, which then calls `Collateral.settleProduct()`. The only restriction on the `updateClosed()` method is that the sender is the product's coordinator.

Impact Malicious product coordinators can still modify the bookkeeping even if the protocol is paused, which result in harmful effects for users of that product or for the protocol.

Recommendation There are several possible solutions that the developers could adopt, each with different tradeoffs:

- The `notPaused` modifier can be added to `settleAccount()` and `settleProduct()` . This will ensure that settlement cannot occur while the protocol is paused, but at higher gas costs (as the check needs to occur on every call).
- The `notPaused` modifier can be added to the methods in `Product`. This will save some gas compared to the other method, but the developers will need to be careful to ensure that future methods added to `Product` correctly engage with the pausing mechanism.

Lastly, the developers should add a unit test to ensure that `Product.updateClosed()` cannot be called when the protocol is paused.

Developer Response The developers added `notPaused` to `Product.updateClosed()`.

4.1.10 V-VUL-PER-010: Product owner can DoS new or inactive users by creating many incentive programs

Severity	Low	**Commit**	d579241
Type	Denial of Service	**Status**	Won't Fix
File(s)	`Incentivizer.sol`		
Location(s)	create(), syncAccount()		

The protocol owner or a product coordinator can use `Incentivizer.create()` to create an incentive program that awards a caller-specified ERC20 token to makers and takers over time. As part of the bookkeeping, this will eventually lead to a call to `ProductManagerLib.register()`, which will add the newly created program to the `programInfos` array. Whenever the product's

```
function register(
        ProductManager storage self,
        ProgramInfo memory programInfo
  ) internal returns (uint256 programId) {
      programId = self.programInfos.length;
      self.programInfos.push(programInfo);
      self.programs[programId].initialize(programInfo);
      self.activePrograms.add(programId);
  }
```

Snippet 4.13: Definition of `ProductManager.register()`

settlement logic is run for a given user in `Product._settleAccount()`, the reward program bookkeeping will be updated in `ProductManagerLib.syncAccount()` . Part of this logic is to loop over all "unseen" programs since the user's account was last settled. However, if the user is entirely new to the product, or a large number of programs have been added since the user last settled their account, then the starting index `fromProgramId` will be small. Because the ending index `toProgramId` is the number of programs that have been created, the loop could iterate over all reward programs. This will result in a large gas cost, especially because one or more storage writes will be performed on each iteration. Consequently, there may be a denial-of-service issue where `syncAccount()` reverts or becomes too expensive to call.

```
function syncAccount(
      ProductManager storage self,
      IProduct product,
      address account,
      IOracleProvider.OracleVersion memory currentOracleVersion
) internal {

  // Add any unseen programs
  uint256 fromProgramId = self.nextProgramFor[account];
  uint256 toProgramId = self.programInfos.length;
  for (uint256 programId = fromProgramId; programId < toProgramId; programId++) {
      self.activeProgramsFor[account].add(programId);
  }
```

Snippet 4.14: Lines in `ProductManagerLib.syncAccount()`, where the looping occurs.

Impact Because individual account settlement is performed on any user-facing flow, and individual account settlement always updates the reward program bookkeeping, a revert there could cause user-facing flows to revert. This can occur if the user has not interacted with the protocol in a long time and the product coordinator has created many reward programs.

Note that there is a configuration parameter limiting the maximum number of incomplete reward programs that may run for any given product (regardless of whether they have been started or not), which mitigates this issue. However, the parameter can be changed by the protocol owner at any time.

Recommendation To reduce the possibility of this issue occurring, the developers could investigate ways to avoid redundant calls to `activeProgramsFor[account].add(programId)`.

Developer Response The developers do not plan to fix this for the following reason:

> Unfortunately we don't have a very good way of resolving this without doing a large refactor. We could special case to "skip" completed programs for accounts with position == 0, but this wouldn't resolve the overall problem. The coordinator could still open many quick programs while a user has a position open, and all would need to be run through for proper accounting.

4.1.11 V-VUL-PER-011: Unspecified Solidity compiler behavior in MultiInvokerRollup

Severity	Low	**Commit**	d579241
Type	Maintainability	**Status**	Fixed
File(s)	`MultiInvokerRollup.sol`		
Location(s)	_decodeFallbackAndInvoke()		

The `MultiInvokerRollup` is designed to implement an optimized encoding scheme for batched calls to Perennial contracts. However, the call arguments are decoded in a way that involves unspecified behavior in Solidity. Specifically, in each case in `_decodeFallbackAndInvoke()`, all of the side-effecting "_read" functions used to parse the arguments are called in the same tuple assignment statement. The Solidity compiler documentation mentions that evaluation order of

```
(address account, address product, UFixed18 amount) =
                    (_readAndCacheAddress(input, ptr), _readAndCacheAddress(input,
    ptr), _readUFixed18(input, ptr));
```

Snippet 4.15: Example snippet in `MultiInvokerRollup` where unspecified behavior occurs.

expressions is not specified:

> The evaluation order of expressions is not specified (more formally, the order in which the children of one node in the expression tree are evaluated is not specified, but they are of course evaluated before the node itself). It is only guaranteed that statements are executed in order and short-circuiting for boolean expressions is done.

Impact The Solidity compiler is free to reorder the "_read" calls, which may result in subtle errors in the way that the call arguments are decoded. The ordering is not guaranteed to be stable across Solidity versions or even different Yul optimizer settings.

Recommendation To enforce evaluation order, the developers should split each tuple assignment into individual assignment statements.

4.1.12 V-VUL-PER-012: Potential selector collision in MultiInvokerRollup encoding scheme

Severity	Low	**Commit**	d579241
Type	Hash Collision	**Status**	Fixed
File(s)	`MultiInvokerRollup.sol`		
Location(s)	_decodeFallbackAndInvoke()		

In order to reduce the size of calldata, the `MultiInvokerRollup` contract provides a custom call argument encoding for calls to `MultiInvoker`. This is implemented through the fallback function, which passes the raw call data to `_decodeFallbackAndInvoke()`. However, the encoding scheme makes it possible for the encoded call arguments to collide with actual external function selectors that are handled by the `MultiInvokerRollup`.

Impact A user may make a legitimate call to the `MultiInvokerRollup` contract with the intention of activating the fallback function. However, if the first four bytes of the encoded call data collides with a function selector, an external function may be invoked. This may lead to the execution of a successful transaction that a user does not intend to actually execute.

Recommendation To avoid this problem entirely, the developers can modify their encoding scheme so that the first byte (or first two bytes) must match a "magic number" that is guaranteed to not match the first byte of any other selector, thereby preventing a selector collision. The remaining call data can be parsed using the original encoding scheme.

4.1.13 V-VUL-PER-013: MultiInvoker withdraw & unwrap reverts when withdrawing max

Severity	Low	**Commit**	d579241
Type	Logic Error	**Status**	Fixed
File(s)	`MultiInvoker.sol`		
Location(s)	_withdrawAndUnwrap()		

The `MultiInvoker` contract allows users to batch calls to the Perennial protocol contracts. One such call is the `_withdrawAndUnwrap()` functionality, which will call `Collateral.withdrawFrom()` to withdraw DSU from the protocol and then convert it to USDC. However, this does not correctly handle the case where `amount` can be set to `type(uint256).max`: the `Collateral` contract will treat such an `amount` as withdrawing all deposited DSU, but the `_handleUnwrap()` method will attempt to swap `type(uint256).max` amount of DSU to USDC, leading to a revert.

```
function _withdrawAndUnwrap(address receiver, IProduct product, UFixed18 amount)
    internal {
    // Withdraw the amount from the collateral account
    collateral.withdrawFrom(msg.sender, address(this), product, amount);

    _handleUnwrap(receiver, amount);
}

function _handleUnwrap(address receiver, UFixed18 amount) internal {
    // If the batcher is 0 or doesn't have enough for this unwrap, go directly to the
     reserve
    if (address(batcher) == address(0) || amount.gt(USDC.balanceOf(address(batcher)))
    ) {
        reserve.redeem(amount);
        if (receiver != address(this)) USDC.push(receiver, amount);
    } else {
        // Unwrap the DSU into USDC and return to the receiver
        batcher.unwrap(amount, receiver);
    }
}
```

Snippet 4.16: Definitions of `_withdrawAndUnwrap()` and `_handleUnwrap()`

Impact The `_withdrawAndUnwrap()` may revert unexpectedly when passed a legitimate amount. Users may believe they are being "locked out" of their funds when they try to withdraw "everything", which could result in reputational damage.

Recommendation The `Collateral.withdrawFrom()` method should be changed to return the amount that is actually withdrawn. This actual amount can then be passed to `_handleUnwrap()`.

Developer Response For the case of withdrawing the uint256 max, the developers fixed this bug by adding logic to obtain the account's actual DSU balance before withdrawing it and converting to USDC.


```
function withdrawFrom(address account, address receiver, IProduct product, UFixed18
    amount)
public
/* ... long list of modifiers ... */
{
        amount = amount.eq(UFixed18Lib.MAX) ? collateral(account, product) : amount;
        _products[product].debitAccount(account, amount);
        token.push(receiver, amount);
    emit Withdrawal(account, product, amount);
}
```

Snippet 4.17: Relevant lines in `Collateral.withdrawFrom()`

4.1.14 V-VUL-PER-014: Missing parameter validation in BalancedVault constructor

Severity	Low	**Commit**	d579241
Type	Data Validation	**Status**	Fixed
File(s)	`BalancedVaultDefinitino.sol`		
Location(s)	constructor()		

The constructor of the `BalancedVaultDefinition`, which the `BalancedVault` contract inherits from, is missing several crucial parameter checks. Notably:

- the `asset` address is assumed to be the same as the `collateral`'s token, but the deployer of the vault can set `asset` to anything.
- a vault can be created with no markets, which does not seem very useful
- there are no checks for whether the products in each market definition are actually valid products
- weights can be set to 0
- `targetLeverage` can be set to 0

Impact

- The vault deployer may be able to set `asset` to a malicious smart contract, which could be used to execute reentrancy attacks.
- Even if the `asset` is not a malicious smart contract and is instead a different token, then a call to `deposit()` may still succeed if the amount of deposited assets is less than the `Controller.minCollateral()`. In this case, the `_rebalance()` call in `deposit()` will not attempt to deposit funds into the `Collateral` contract (which would otherwise revert because the vault would not be approved to transfer collateral tokens to the `Collateral` contract). Furthermore, the bookkeeping would be incorrect during settlement, which could result in `redeem()` or `claim()` reverting and therefore a locked funds issue.
- The vault deployer could set one or more products to a malicious contract controlled by the attacker. If the malicious contract is able to bypass the collateral rebalancing logic, it may be able to execute reentrancy attacks through calls to `openMake()` and `closeMake()` methods during position rebalancing.
- If there are no markets, or weights or `targetLeverage` are set to zero, then this will result in the vault being less useful than it should be. For example, if `targetLeverage` is zero, then the vault will not open any positions, as the target number of positions will be zero when rebalancing.

Recommendation The developers should, at minimum, add the following checks to the constructor:

- There is at least one market
- All weights are greater than zero
- All long and short markets are valid products (by using `Controller.isProduct()`)
- `targetLeverage` is greater than zero

Furthermore, they should change `asset` so that it is assigned from `collateral.token()` rather than allowing the sender to specify it as a parameter.

4.1.15 V-VUL-PER-015: BalancedVault does not initialize oracle version of first market

Severity	Low	**Commit**	d579241
Type	Logic Error	**Status**	Fixed
File(s)	`BalancedVault.sol`		
Location(s)	initialize()		

In `initialize()`, there is a loop that will set the oracle version of each market to the last settled oracle version of the long product. However, the loop starts at index 1 rather than index 0, which means that the oracle version for index 0 may not be initialized correctly.

```
// Stamp new market's data for first epoch
(EpochContext memory context, )  = _settle(address(0));
for (uint256 marketId = 1; marketId < totalMarkets; marketId++) {
    if (_marketAccounts[marketId].versionOf[context.epoch] == 0) {
        _marketAccounts[marketId].versionOf[context.epoch] = markets(marketId).long.
    latestVersion();
    }
}
```

Impact The oracle version of a market is used for checking whether a vault's epoch is stale or complete, so this issue could affect the core bookkeeping in the vault.

Recommendation The developer should confirm why they set the index to start at 1, and change it to 0 if this results in an issue with the bookkeeping.

Developer Response The developers change the index to start at 0 and added code comments explaining why the loop is needed.

4.1.16 V-VUL-PER-016: Inconsistent Solidity compiler versions

Severity	Warning	**Commit**	d579241
Type	Maintainability	**Status**	Fixed
File(s)		N/A	
Location(s)		N/A	

The version of the Solidity compiler is inconsistent across several contracts, and in some places, is required to be a specific version. Specifically, most of the contracts use Solidity 0.8.15, while some contracts such as `Product` use Solidity 0.8.17. This may result in unexpected minor variations in behavior in the Solidity code.

Recommendation Use the same Solidity compiler version pragma in all source files to avoid potential behavior differences due to differing Solidity compiler versions.

Developer Response The developers noted that this difference is because they deployed an initial version of the contracts that used Solidity 0.8.15, and that they moved some of the contracts to 0.8.17 to reduce the code size.

4.1.17 V-VUL-PER-017: Risk of encountering known Solidity compiler bugs

Severity	Warning	**Commit**	d579241
Type	Maintainability	**Status**	Fixed
File(s)		N/A	
Location(s)		N/A	

Several source files require Solidity compiler versions to be strictly equal to 0.8.15, which can suffer from the following known Solidity compiler bugs:

- https://blog.soliditylang.org/2022/08/08/calldata-tuple-reencoding-head-overflow-bug/
- https://blog.soliditylang.org/2022/09/08/storage-write-removal-before-conditional-termination/

Impact

- The `Controller.createProduct()` method (which is compiled with Solidity 0.8.15) uses `abi.encodeCall(...)` with a calldata struct parameter. While this is not at risk of the "Calldata Tuple Reencoding Head Overflow Bug", a future change to the `productInfo` parameter or the struct may trigger the bug.
- Because the majority of the source code is compiled with Solidity 0.8.15 with the Yul optimizer enabled, there is an increased chance of triggering the "Store Write Removal Before Conditional Termination" bug.

Recommendation Upgrade to at least Solidity 0.8.17 to avoid the known compiler bugs mentioned above. If the developers perform this upgrade, they should test their protocol to ensure that it is not affected by ***unknown*** bugs in the new compiler version.

4.1.18 V-VUL-PER-018: Invalid product parameter defaults to protocol owner

Severity	Warning	Commit	d579241
Type	Data Validation	Status	Fixed
File(s)	`Controller.sol`		
Location(s)	treasury(IProduct), owner(IProduct)		

The `Controller` contract providers two methods `treasury()` and `owner()` to query the product treasury address and product owner, respectively. However, if a caller provides an invalid product address, then `coordinatorFor[product]` will default to zero. This means that for an invalid product, `treasury()` and `owner()` will return the treasury and owner of the protocol, respectively.

```
function treasury(IProduct product) external view returns (address) {
    return treasury(coordinatorFor[product]);
}
```

Snippet 4.18: Definition of `treasury(IProduct)`

```
function owner(IProduct product) external view returns (address) {
    return owner(coordinatorFor[product]);
}
```

Snippet 4.19: Definition of `owner(IProduct)`

Impact If callers end up calling either of the aforementioned method with an invalid `Product`, they may consequently send fees to the wrong address (using `treasury()`) or authorize users that should not be authorized (using `owner()`).

Recommendation The methods should revert if the product is invalid (which occurs when `coordinatorFor[product] == 0`). Callers must call `isProduct(product)` to correctly handle this case.

4.1.19 V-VUL-PER-019: Dangerous memory aliasing in deductFee

Severity	Warning	Commit	d579241
Type	Maintainability	Status	Fixed
File(s)	`Incentivizer.sol`,`ProgramInfo.sol`		
Location(s)	Incentivizer.create(), ProgramInfo.deductFee()		

A product coordinator or the protocol owner can create an incentive program by calling `Incentivizer.create()`. As part of this, the protocol will detect a percentage fee from the `amount` of tokens will be offered as part of the incentive program. The fee deduction is calculated in `ProgramInfoLib.deductFee()`, which takes a memory reference to a `ProgramInfo` struct. The way

```
function deductFee(ProgramInfo memory programInfo, UFixed18 incentivizationFee)
internal pure returns (ProgramInfo memory, UFixed18) {
    Position memory newProgramAmount = programInfo.amount.mul(UFixed18Lib.ONE.sub(
    incentivizationFee));
    UFixed18 programFeeAmount = programInfo.amount.sub(newProgramAmount).sum();
    programInfo.amount = newProgramAmount;
    return (programInfo, programFeeAmount);
}
```

Snippet 4.20: Implementation of `deductFee()`

that the fee is calculated is error-prone. First, the assignment `programInfo.amount` is actually modifying the original `programInfo` struct. Second, the original `programInfo` is being returned from this function, which means that the original `programInfo` and the returned value both point to the same struct (i.e., they alias). If the caller of `deductFee()` uses `programInfo` after the call to `deductFee()`, they may not realize that the `amount` field has been changed. This can lead to bugs being introduced.

Impact Currently, we do not believe this has an impact; however, it could lead to maintainability problems in the future. The only caller of `deductFee()` is `Incentivizer.create()`, which passes a *calldata* `ProgramInfo` to `deductFee()`. This means that the calldata `ProgramInfo` will first be copied into a newly allocated struct that is then passed into `deductFee()`. Thus, the struct returned from `deductFee()` will not alias with the calldata `ProgramInfo`. However, if `Incentivizer.create()` is refactored to assign the calldata `ProgramInfo` into a memory variable, then it is possible for the aliasing issue occur. This could result in the amount of tokens transferred from the sender to be lower than expected, making the actual token amounts inconsistent with the bookkeeping.

Recommendation `deductFee()` should either:

- Create a copy of the `programInfo` argument, assign to the copy, and then return the copy; or
- Remove the `ProgramInfo` return value and explicitly document that the `programInfo` argument's `amount` field will be updated.

Developer Response The developers updated the code according to the second recommendation above.

```
function create(IProduct product, ProgramInfo calldata programInfo)
external
/* ... long list of modifiers ... */
returns (uint256 programId) {
    /* ... parameter validation ... */

    // Take fe
    (ProgramInfo memory newProgramInfo, UFixed18 programFeeAmount) = ProgramInfoLib.
    deductFee(programInfo, _controller.incentivizationFee());
    fees[newProgramInfo.token] = fees[newProgramInfo.token].add(programFeeAmount);

    // Register program
    programId = _products[product].register(newProgramInfo);

    // Charge creator
    newProgramInfo.token.pull(msg.sender, programInfo.amount.sum());
```

Snippet 4.21: Relevant lines in `Incentivizer.create()`

4.1.20 V-VUL-PER-020: Error-prone code duplication in MultiInvokerRollup

Severity	Warning	**Commit**	d579241
Type	Maintainability	**Status**	Fixed
File(s)	`MultiInvokerRollup.sol`		
Location(s)	_decodeFallbackAndInvoke()		

In `_decodeFallbackAndInvoke()`, the `action` is decoded from the first byte, and then an appropriate function is called based on the value of the action. However, each case compares the action to a hardcoded integer constant, unlike in `MultiInvoker`. This is error-prone because a developer making changes to the `PerennialAction` enum may forget to also make the corresponding changes in `MultiInvokerRollup`. The `MultiInvokerRollup` should instead compare the values of `action` to values of `PerennialAction`.

4.1.21 V-VUL-PER-021: Significant gas costs due to MultiInvokerRollup address cache

Severity	Warning	Commit	d579241
Type	Gas Optimization	Status	Won't Fix
File(s)	`MultiInvokerRollup.sol`		
Location(s)	_readAndCacheAddress()		

As part its encoding scheme, the `MultiInvokerRollup` contract uses a run-length encoding for address data. An address is represented using a pair `(len, data)` where `len` is a byte. If `len` is zero, the `data` is a 160-bit integer (i.e., an Ethereum address), and it will be inserted into an "address cache" storage data structure. If `len` is nonzero, then the `data` is a length `len` byte array; the address will be retrieved from the address cache by looking up the entry of the address cache at `data`. This allows a saving of up to 18 bytes compared to directly encoding an address in the call data.

While this encoding scheme can reduce the size of call data, it is very expensive in terms of gas: the address cache store is implemented in terms of three storage writes (two from appending to a storage array, and one from writing to a mapping), for a total of over 40000 gas for a new cache entry (three `SSTORE` instructions, two of which are to a zero-valued entry). An address cache lookup will cost at least 2100 gas (cold `SLOAD` instruction).

Impact The additional storage costs of the address cache may defeat the purpose of the encoding scheme, in which case the address cache brings additional complexity and maintenance costs for no benefit. Furthermore, the address cache requires users to remember what indices previously cached addresses were stored at, which can be error-prone.

Recommendation The developers should compare the gas costs of the storage operations and the gas savings of the address encoding scheme for what they believe will be typical usage scenarios. This will help determine whether the address cache is effective in saving gas.

Developer Response The developers responded:

> The SSTOREs used here are all required, and while unknown addresses will incur an extra cost, after they are known the savings will quickly offset the costs after about 2-3 reads.

The developers also noted that they plan to run their protocol on several layer-2 solutions where gas costs scale with the size of the call data. They expect users to "populate" the cache with well-known addresses beforehand in order, so insertions into the address cache will be less frequent than reads.

4.1.22 V-VUL-PER-022: Double check non-negative price assumption

Severity	Warning	**Commit**	d579241
Type	Maintainability	**Status**	Fixed
File(s)	`perennial-oracle/contracts`		
Location(s)	N/A		

Several comments in the `perennial-oracle` package suggest that only non-negative prices are supported. However, after a discussion with the developers, the developers noted that it was not clear whether such a restriction is required. This issue serves as reminder to the developers to check whether non-negative prices should be supported.

Developer Response The developers removed the comment; negative prices are supported.

4.1.23 V-VUL-PER-023: No validation in Incentivizer.owner

Severity	Warning	Commit	d579241
Type	Data Validation	Status	Fixed
File(s)	`Incentivizer.sol`		
Location(s)	owner()		

Similar to V-VUL-PER-018, the `Incentivizer.owner()` method will return the protocol owner if it is passed an invalid protocol address or an invalid program ID, as the final coordinator ID passed to `Controller.owner()` will be 0.

```
1 function owner(IProduct product, uint256 programId) public view returns (address) {
2     return controller().owner(_products[product].programInfos[programId].
      coordinatorId);
3 }
```

Snippet 4.22: Definition of `owner()`

4.1.24 V-VUL-PER-024: BalancedVault allows deposit to address 0

Severity	Warning	**Commit**	d579241
Type	Maintainability	**Status**	Won't Fix
File(s)	`BalancedVault.sol`		
Location(s)	deposit()		

The `BalancedVault.deposit()` method allows users to deposit funds for the zero address. The zero address is treated specially by the contract and is used for global bookkeeping.

Impact While the auditors have not identified any obvious negative effects from such calls in the current version of the `BalancedVault` , it may be possible that future changes to the `BalancedVault` may assume the zero address cannot receive deposits.

Recommendation To guard against future errors, we recommend that the developers revert if `deposit()` is called with a zero address.

Developer Response The developers stated that the frontend application to the vaults will block deposits to the zero address, and that they "won't be fixing this as we prefer not to prevent user error unless it will cause logic issues within the contract."

4.1.25 V-VUL-PER-025: No checks against oracle updated timestamp of zero

Severity	Warning	**Commit**	d579241
Type	Data Validation	**Status**	Fixed
File(s)	`ChainlinkAggregator.sol`		
Location(s)	See description		

The `ChainlinkAggregatorLib.getLatestRound()` function is a wrapper around the `getLatestRound()` method on a chainlink aggregator, returning the latest round ID, update timestamp, and price. This function is mainly called from the `sync()` method of `ChainlinkOracle` and `ChainlinkFeedOracle`, where the update timestamp is passed through as-is as part of the return value of `sync()` without validation. The timestamp is then used in the following places:

- To calculate the elapsed time when computing accumulated funding in `VersionedAccumulatorLib._accumulateFunding()`
- To calculate the elapsed time when computing accumulated shares for reward programs in `VersionedAccumulatorLib._accumulateShare()`
- To determine the start and complete time of reward programs.
- In `ChainlinkAggregatorLib.getPhaseSwitchoverData()` to determine when the phase of a Chainlink feed proxy changes.

However, there are no checks that the timestamp is nonzero within the core protocol. According to the Chainlink documentation:

> A read can revert if the caller is requesting the details of a round that was invalid or has not yet been answered. If you are deriving a round ID without having observed it before, the round might not be complete. To check the round, validate that the timestamp on that round is not 0.

Thus, if the oracle reports an update timestamp of zero, it could cause errors in the core protocol.

Impact The auditors were unable to confirm this issue in detail, as the Chainlink documentation is vague on when exactly the timestamp can be zero. When asked, the Perennial developers also said that are unsure whether zero updated-at timestamps can still occur.

Regardless, if an update timestamp can be set to zero, then settlement is likely to revert due to subtraction overflow: the "current" oracle version will have a timestamp of zero, so when calculating elapsed time between two consecutive oracle versions, the contract will attempt to subtract a nonzero number from zero.

Recommendation The developers should try to determine if zero update timestamps can occur in practice, and if so, they should insert checks in the oracle `sync()` methods to handle zero timestamps.

4.1.26 V-VUL-PER-026: No check that starting round ID is valid

Severity	Warning	**Commit**	d579241
Type	Data Validation	**Status**	Acknowledged
File(s)	`ChainlinkOracle.sol`		
Location(s)	constructor()		

When the `ChainlinkOracle` is initialized, it inserts two entries into the `_phases` array: a dummy entry for index 0, and an actual entry for phase 1. In particular, the phase 1 entry sets the starting oracle version to 0 and the starting round ID to the first round ID of the first phase. However, if the base/quote pair is invalid, then it may be possible for `registry_.getStartingRoundId()` to return 0.

```
// phaseId is 1-indexed, skip index 0
_phases.push(Phase(uint128(0), uint128(0)));
// phaseId is 1-indexed, first phase starts as version 0
_phases.push(Phase(uint128(0), uint128(registry_.getStartingRoundId(base_, quote_, 1)
    )));
```

Snippet 4.23: Relevant lines in the constructor

Recommendation The constructor should revert if the starting round ID is 0, as that would indicate that the base/quote pair is invalid.

Developer Response The developers acknowledged the issue and stated:

> We'll opt to not fix this in Solidity and instead add checks to our deploy and verification scripts

4.1.27 V-VUL-PER-027: Error-prone initialization in constructor of upgradable contracts

Severity	Warning	Commit	d579241
Type	Maintainability	Status	Fixed
File(s)	`BalancedVaultDefinition.sol`		
Location(s)	constructor()		

The `BalancedVault` is an upgradable contract which inherits from the `BalancedVaultDefinition` contract. The latter appears to be mostly used for configuration, and all of its storage variables are immutable. Because the storage variables are immutable, they are directly inlined into the

```
IController public immutable controller;
ICollateral public immutable collateral;
UFixed18 public immutable targetLeverage;
```

Snippet 4.24: Some of the storage variables in `BalancedVaultDefinition`

logic contract at deployment time. Due to this, all the further upgraded versions of the contract have to be deployed with exactly the same constructor arguments as the first version deployed, or else the behavior of the new version may deviate from the behavior of the old version.

Furthermore, there is no logic in either `BalancedVault` or `BalancedVaultDefinition` that checks whether an upgraded version will have the same parameters as the previous version.

Impact If the immutable variables are changed between upgrades, this may corrupt the bookkeeping in the vault. For example, if the deployer accidentally changes the collateral contract, then the bookkeeping will become inconsistent.

Based on a discussion with the developers, the vaults are intended to be deployed by product coordinators; and in practice, they can be deployed by anyone. It is likely that these deployers may run into the issue described above.

Recommendation To reduce the likelihood of deployment errors occurring, the developers should classify the immutable variables into two categories: "constant" and "configurable". The "constant" variables should remain the same between upgrades. For example, the Collateral and Controller addresses should remain the same, and existing markets should retain the same long/short product pairs. The "configurable" variables are those that allowed to be dynamically modified, such as the weight of each market.

We note that the pattern of using immutable variables in upgradable contracts is used throughout the protocol. The developers should review their other upgradable contracts for similar error-prone patterns.

Developer Response The developers added an optional "previous implementation" parameter to the BalancedVault constructor, which is used to validate that some of the "constant" variables we have described above are not changed between upgrades.

4.1.28 V-VUL-PER-028: ReservoirFeedOracle does not validate starting round

Severity	Warning	Commit	d579241
Type	Data Validation	Status	Fixed
File(s)	`ReservoirFeedOracle.sol`		
Location(s)	constructor()		

Similar to V-VUL-PER-026, the constructor of `ReservoirFeedOracle` does not validate that the `_versionOffset` parameter corresponds to a valid round.

Developer Response The developers noted that they no plans to deploy the `ReservoirFeedOracle`, and they have removed it from the code base.

4.1.29 V-VUL-PER-029: getStartingRoundId assumes phase is invalid if first round is invalid

Severity	Warning	Commit	d579241
Type	Data Validation	Status	Acknowledged
File(s)	`ChainlinkAggregator.sol`		
Location(s)	getStartingRoundId()		

`ChainlinkAggregator.getStartingRoundId()` is a helper function used to retrieve the first valid round in a phase of a Chainlink aggregator proxy. It calls the `getRoundData()` method with the first aggregator round ID of that phase. According to the Chainlink documentation, if the retrieved round data has an update timestamp of zero, then it is an incomplete or invalid round. However, the code assumes that if the first round is invalid, then the entire phase is invalid. This is not necessarily true, since there may be rounds ***after*** the first round that have nonzero update timestamps.

```
1 function getStartingRoundId(ChainlinkAggregator self, uint16 phaseId, uint256
      targetTimestamp)
2 internal view returns (uint256) {
3     AggregatorProxyInterface proxy = AggregatorProxyInterface(ChainlinkAggregator.
      unwrap(self));
4
5     (,,,uint256 startTimestamp,) = proxy.getRoundData(uint80(
      _aggregatorRoundIdToProxyRoundId(phaseId, 1)));
6     if (startTimestamp == 0) return 0; // Empty phase
7
8     return _search(proxy, phaseId, targetTimestamp, startTimestamp, 1);
9 }
```

Snippet 4.25: Relevant lines in `getStartingRoundId()`

Developer Response The developer's understanding of the Chainlink API is that aggregator proxies no longer return update timestamps of 0; however, they agree that the assumption would be incorrect if the first round could have an update timestamp of 0. They note that there is no easy way to confirm if update timestamps of 0 could occur.

4.1.30 V-VUL-PER-030: Shares are accumulated when product is closed

Severity	Warning	**Commit**	d579241
Type	Logic Error	**Status**	Intended Behavior
File(s)	`VersionedAccumulator.sol`		
Location(s)	_accumulateShare()		

The `VersionedAccumulatorLib._accumulateShare()` function is used to calculate the number of tokens per position that should be awarded to users of a product for a reward program between two oracle versions. Unlike in other functions such as `_accumulateFunding()` and `_accumulatePosition()`, shares will still be accumulated even when the product is closed. The developer should confirm whether this is intended behavior.

Developer Response The developers responded:

> This is intended behavior, while this isn't ideal it is necessary for the incentivizer accounting to be correct. If shares aren't accumulated, then the incentivizer bookkeeping will be inaccurate.

4.1.31 V-VUL-PER-031: Potential extra memory copy in _claimProduct

Severity	Info	**Commit**	d579241
Type	Gas Optimization	**Status**	Fixed
File(s)	`Incentivizer.sol`		
Location(s)	See description		

The `_claimProduct()` method takes a memory array of program IDs as its third argument. However, at all of its call sites, it is being passed a calldata array. Thus, it may be possible that the Solidity compiler is inserting an extra memory copy before calling `_claimProduct()`. It may be possible to save a small amount of gas by declaring the `programIds` argument of `_claimProduct()` as `calldata`.

```
1 function _claimProduct(address account, IProduct product, uint256[] memory programIds
      )
```

Snippet 4.26: The function parameters of `_claimProduct()`

```
function claim(IProduct product, uint256[] calldata programIds)
external
nonReentrant
{
    _claimProduct(msg.sender, product, programIds);
}

function claimFor(address account, IProduct product, uint256[] calldata programIds)
external
nonReentrant
onlyAccountOrMultiInvoker(account)
{
    _claimProduct(account, product, programIds);
}

function claim(IProduct[] calldata products, uint256[][] calldata programIds)
external
nonReentrant
{
    if (products.length != programIds.length) revert
    IncentivizerBatchClaimArgumentMismatchError();
    for (uint256 i; i < products.length; i++) {
        _claimProduct(msg.sender, products[i], programIds[i]);
    }
}
```

Snippet 4.27: The functions that invoke `_claimProduct()`

4.1.32 V-VUL-PER-032: Unnecessary memory allocations in MultiInvokerRollup

Severity	Info	Commit	d579241
Type	Gas Optimization	Status	Fixed
File(s)	`MultiInvokerRollup.sol`		
Location(s)	_readUint8(), _readUint256()		

When the `MultiInvokerRollup` contract decodes the call arguments, it uses the utility functions `_bytesToUint8()`, `_bytesToAddress()`, and `_bytesToUint256()` to decode the arguments (via an `MLOAD` inline assembly instruction). However, because the call arguments are stored in `calldata` and are passed to functions that take `memory` arguments, this means that the call arguments are actually being copied to newly allocated memory data each time, which are then read. This incurs unnecessary gas costs.

Recommendation The functions named above should take a `bytes calldata` as an argument and directly load from the data using a `calldataload` instruction. For example, the developers may want to consider an implementation such as the one below. This can avoid extra gas fees from both memory allocation and memory load operations, which may scale superlinearly past a threshold.

However, use of inline assembly is more error-prone than plain Solidity, so we recommend that the developers thoroughly test their code if they intend to make such optimizations.

4.1.33 V-VUL-PER-033: Settlement assumes contiguity of oracle versions

Severity	Info	**Commit**	d579241
Type	Logic Error	**Status**	Intended Behavior
File(s)	See description		
Location(s)	See description		

To determine the oracle version at which settlement can occur, the `Product._settle()` and `Product._settleAccount()` methods call `PrePositionLib.settleVersion()`. This works as follows: if the settled oracle version of the `PrePosition` (i.e., unsettled maker/taker orders) is 0 (i.e., no orders placed), then the version to settle is the current oracle version. Otherwise, the version to settle (i.e., to apply the orders to) is the oracle version immediately after the current one (which may or may not exist yet). This means that the settlement logic assumes that oracle versions are contiguous, with no "gaps" between oracle versions. However, based on the implementations in

```
function settleVersion(PrePosition storage self, uint256 currentVersion) internal
    view returns (uint256) {
    uint256 _oracleVersion = self.oracleVersion;
    return _oracleVersion == 0 ? currentVersion : _oracleVersion + 1;
}
```

Snippet 4.28: Definition of `PrePositionLib.settleVersion()`

```
uint256 _settleVersion = _position.pre.settleVersion(currentOracleVersion.version);
IOracleProvider.OracleVersion memory settleOracleVersion = _settleVersion ==
    currentOracleVersion.version
    ? currentOracleVersion // if b == c, don't re-call provider for oracle version
    : atVersion(_settleVersion);
```

Snippet 4.29: How the settled version is determined.

`ChainlinkOracle.sync()` and `ChainlinkFeedOracle.sync()`, the oracle versions may not necessary be contiguous. Specifically, each of the above methods eventually lead into a call to an internal `_buildOracleVersion()` function, which calculates the oracle versions as follows:

```
uint256(phase.startingVersion) + round.roundId - uint256(phase.startingRoundId)
```

In prose: given a Chainlink round ID, an oracle version is calculated as the oracle version at the start of the round's phase plus the number of rounds since the beginning of that phase.

Impact The Chainlink round IDs need not be contiguous; therefore the oracle versions may not be contiguous. It may be possible for the settlement version to correspond to an invalid oracle `roundId`, which could be inconsistent with the assumptions used by the protocol's bookkeeping. An invalid round may report stale price information, provide an update timestamp of zero, or produce other invalid information that the protocol currently does not guard against.

Developer Response The developers responded:

We assume contiguity of rounds within a Chainlink phase, and handle phase changes within the sync function to ensure the non-contiguity of rounds still results in contiguity of versions. We cannot find any information within Chainlink's docs that state that a round ID can be invalid within a phase if there is a confirmed valid round after that round ID (in the same phase).