Zellic

Ionic Protocol

Smart Contract Security Assessment

May 19, 2022

Prepared for:

Rahul Sethuram

Ionic Protocol

Prepared by:

Chad McDonald and Katerina Belotskaia

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.

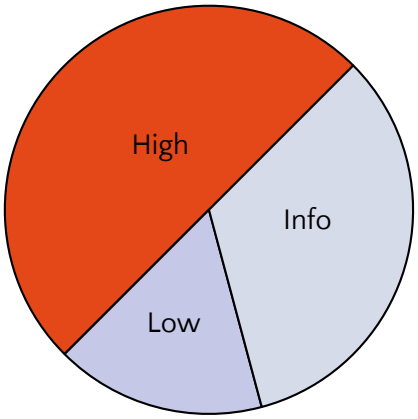
1 Executive Summary

Zellic conducted an audit for Ionic Protocol from April 26th to May 7th, 2022 on the scoped contracts and discovered 6 findings. The focus of the audit was the modified Fuse contracts along with oracle implementations, liquidation strategies for wrapped/deposited tokens and the custom ERC4626 strategies which wrap yield-bearing protocols as specified by the client. Fortunately, no critical issues were found. The audit uncovered 2 findings of medium impact, 2 of low impact and 2 informational. Additionally, Zellic recorded its notes and observations from the audit for Ionic Protocol's benefit at the end of the document.

Our general overview of the code is that the codebase is still in active development and as such there are improvements in testing, documentation and readability that have yet to be implemented. Code documentation in particular could be expanded upon to make the developers intentions clear and reduce confusion for future auditors, developers and users.

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	3
Medium	0
Low	1
Informational	2



2 Introduction

2.1 About Ionic Protocol

Ionic Protocol is building a cross chain app that will be connecting EVM chains through Fuse pools.

Fuse is an open interest rate protocol built by Rari Capital that allows users to lend and borrow digital assets. The Fuse protocol enables anyone to instantly create and deploy their own isolated lending and borrowing pool with custom parameters such as provided assets, oracles, administration fees etc.

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 the contracts' 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.

2.3 Scope

The engagement involved a review of the following targets:

Ionic Protocol Contracts

Repository https://github.com/Midas-Protocol/contracts

Version contracts: `9f1f5cfce9667f702b20f80d7928fc8ad86c3274`

Programs

- FuseFeeDistributor.sol
- FusePoolDirectory.sol
- FusePoolLens.sol
- FusePoolLensSecondary.sol
- FuseSafeLiquidator.sol
- ChainlinkPriceOracleV2.sol
- CurveLpTokenPriceOracleNoRegistry.sol
- MasterPriceOracle.sol
- UniswapTwapPriceOracleV2.sol

- BlockVerifier.sol
- MerklePatriciaVerifier.sol
- UniswapOracle.sol
- UQ112x112.sol
- CurveLpTokenLiquidatorNoRegistry.sol
- JarvisSynthereumLiquidator.sol
- XBombLiquidator.sol
- AlpacaERC4626.sol
- AutofarmERC4626.sol
- BeefyERC4626.sol
- BombERC4626.sol
- EllipsisERC4626.sol
- Rlp.sol

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 person weeks. The assessment was conducted over the course of 2 calendar weeks.

Contact Information

The following project managers were associated with the engagement:

Jasraj Bedi, Co-founder
jazzy@zellic.io

Stephen Tong, Co-founder
stephen@zellic.io

The following consultants were engaged to conduct the assessment:

Chad McDonald, Engineer
chad@zellic.io

Katerina Belotskaia, Engineer
kate@zellic.io

2.5 Project Timeline

Zellic conducted an audit for Ionic Protocol from April 26th to May 7th, 2022 on the scoped contracts and discovered 7 findings. Fortunately, no critical issues were found.

The key dates of the engagement are detailed below.

April 26, 2022	Kick-off call
April 26, 2022	Start of primary review period
May 7, 2022	End of primary review period
May 12, 2022	Closing call

3 Detailed Findings

3.1 Unexpected reverts where overflow may be desireable

- **Target**: UniswapTwapPriceOracleV2, UniswapTwapPriceOracleV2Root
- **Category**: Business Logic
- **Severity**: Medium
- **Likelihood**: High
- **Impact**: High

Description

The `UniswapTwapPriceOracleV2` is a modified version of the Compound `UniswapTwapPriceOracleV2` contract. The Compound contract used Open Zeppelin's `SafeMathUpgradeable` to check for arithmetic overflow and underflow issues.

Ionic Protocol removed `SafeMathUpgradeable` and modified the Compound contracts to compile with solidity versions >=0.8.0 which by default includes checked arithmetic to revert on overflows and underflows.

The `UniswapTwapPriceOracleV2` imports `UniswapTwapPriceOracleV2Root` which has also been modified to replace the `SafeMathUpgradeable` functionality with solidity 0.8.0+ default checked arithmetic. However, in `UniswapTwapPriceOracleV2Root` there are portions of code related to price accumulation (`currentPx0Cumu`, `currentPx1Cumu`) and time weighted average price (`price0TWAP`, `price1TWAP`) where arithmetic overflow is desirable. For further reading see Dapp's audit report of Uniswap v2.

This issue was duplicated with a parallel, internal review of the code conducted by Ionic Protocol.

Impact

When calling `getUnderlyingPrice`, an overflow in either `currentPx0Cumu` or `currentPx1Cumu` would lead to an unexpected transaction reversion, rendering the oracle useless.

Recommendations

Review all contracts in the codebase which were updated to compile with solidity 0.8.0+ and place `unchecked` blocks around code where overflow is desireable. This will allow values to wrap on overflows and underflows as expected in versions of solidity prior to 0.8.0.

Below is an example of corrected code for `currentPx0Cumu` in `UniswapTwapPriceOracleV2Root`:

```
function currentPx0Cumu(address pair)
  internal view returns (uint256 px0Cumu) {
  uint32 currTime = uint32(block.timestamp);
  px0Cumu = IUniswapV2Pair(pair).price0CumulativeLast();
  (uint256 reserve0, uint256 reserve1, uint32 lastTime)
  = IUniswapV2Pair(pair).getReserves();
  if (lastTime ≠ block.timestamp) {
    uint32 timeElapsed = currTime - lastTime; // overflow is desired
    px0Cumu += uint256((reserve1 << 112) / reserve0) * timeElapsed;
    unchecked {
      uint32 timeElapsed = currTime - lastTime; // overflow is desired
      px0Cumu += uint256((reserve1 << 112) / reserve0) * timeElapsed;
    }
  }
}
```

Remediation

The issue has been fixed by Ionic Protocol in commit a562fda.

3.2 Improperly set parameter in constructor may lead to failed redemptions

- **Target**: JarvisSynthereumLiquidator
- **Category**: Business Logic
- **Severity**: Medium
- **Likelihood**: Low
- **Impact**: High

Description

Lack of input validation in the `constructor` on the `_txExpirationPeriod` parameter may lead to failed redemptions.

Impact

The variable `txExpirationPeriod` is included as an anti-slippage measure during redemptions as it limits the amount of time a transaction can be included in a block. Mistakenly setting the `_txExpirationPeriod` to 0 or a low value may cause transactions to revert which will block user redemptions.

It is evident from Ionic Protocols' deploy script and tests that they have considered this issue and have appropriately set a `_txExpirationPeriod` time of +40 minutes. Therefore we do not believe this has a security impact presently, but it may lead to future bugs.

Recommendations

Consider including a require statement in the `constructor` to impose a minimum threshold for `_txExpirationPeriod`. The Jarvis documentation recommends setting the expiration period to +30 minutes in the future to account for network congestion.

Remediation

The issue has been fixed by Ionic Protocol in commit 782b54.

3.3 Lack of input validation in initialize

- **Target**: `CurveLpTokenPriceOracleNoRegistry`, `FusePoolLens`
- **Category**: Code Maturity
- **Severity**: Low
- **Likelihood**: Low
- **Impact**: Low

Description

The `initialize` function in both `CurveLpTokenPriceOracleNoRegistry` and `FusePoolLens` does not validate the passed array parameters which may lead to unintended storage outcomes.

Impact

In both of the `initialize` functions, Ionic Protocol uses a for-loop to iterate through array parameters and append them to a mapping variable. If the lengths of the arrays are not equal, the `initialize` call will either revert or complete successfully with missing data.

In `CurveLpTokenPriceOracleNoRegistry`, the mappings `poolOf` and `underlyingTokens` may not be set to the intended values if the length of the array `_lpTokens` is less than the length of either the `_pools` or `_poolUnderlyings` arrays.

In `FusePoolLens`, the mapping variable `hardcoded` stores the mapping of token addresses (`_hardcodedAddresses`) to `TokenData` which includes a token's `name` and `symbol`. If the length of the `_hardcodedAddresses` array is less than the length of the `_hardcodedNames` or `_hardcodedSymbols` arrays, then parameters in those arrays that exist after `_hardcodedAddresses.length` will not be stored.

Recommendations

Consider adding `require` statements in the `initialize` function to validate user-controlled data input and to ensure that array lengths are equal.

Remediation

The issue has been fixed by Ionic Protocol in commit c71037.

3.4 Centralization risk over multiple contracts

- **Target**: Multiple
- **Category**: Code Maturity
- **Likelihood**: Low
- **Severity**: Low
- **Impact**: High

Description

In oracle contracts such as `MasterPriceOracle`, the contract's `admin` has central authority over functions such as `setDefaultOracle`. Likewise in `FusePoolDirectory`, the `admin` has full control over the deployer whitelist.

Impact

In case of a private key compromise, an attacker could change the `defaultOracle` to one which will report a favorable price, sandwiching their swap transaction between two calls to `setDefaultOracle` - the first to set a favorable oracle and the second to return the oracle to the benign default oracle. Similarly, an attacker would be able to whitelist malicious deployer addresses in `FusePoolDirectory`.

Recommendations

- Use a multi-signature address wallet, this would prevent an attacker from causing economic damage if a private key were compromised.
- Set critical functions behind a TimeLock to catch malicious executions in the case of compromise.

Remediation

The issue has been acknowledged by Ionic Protocol and no changes have been made.

Ionic Protocol states, "Before announcing our live platform, we will be transferring admin functionality to MultiSig address, avoiding the risks of single point of failure."

3.5 Remove renounceOwnership functionality

- **Target**: FuseFeeDistributor, FusePoolDirectory and CurveLpTokenPriceOracleNoRegistry
- **Category**: Business Logic
- **Severity**: Informational
- **Likelihood**: N/A
- **Impact**: Informational

Description

The FuseFeeDistributor, FusePoolDirectory and CurveLpTokenPriceOracleNoRegistry contracts implement OwnableUpgradeable which provides a method named `renounceOwnership` that removes the current owner (Reference). This is likely not a desired feature.

Impact

If `renounceOwnership` were called, the contract would be left without an owner.

Recommendations

Override the `renounceOwnership` function:

```
function renounceOwnership() public override onlyOwner{
    revert("This feature is not available.");
}
```

Remediation

Ionic Protocol states that they may remove ownership of the contracts in the future, so the `renounceOwnership` functionality remains. However, they have implemented a two step ownership change pattern for added safety when transferring contract ownership in commit eeea03.

Ionic Protocol states, "in the future we may want to completely remove ownership on the contracts and allow the system to work permissionlessly. All of the contracts are set up to make this possible, so we do not see this as an issue."

3.6 Failing tests and missing coverage in test suite

- **Target**: Multiple contracts
- **Category**: Code Maturity
- **Likelihood**: N/A
- **Severity**: Low
- **Impact**: Informational

Description

Several functions in the smart contracts are not covered by any unit or integration tests, to the best of our knowledge. We ran both the Hardhat test suite and the Forge tests.

In the Hardhat test suite, the tests `createPoolViaSdk` and `createPoolViaContract` both fail.

The Forge tests cover most of the contracts within the scope of this audit. However, the following contracts have no test coverage to the best of our knowledge:

- `FusePoolLens.sol`
- `FusePoolLensSecondary.sol`
- `BlockVerifier.sol`
- `MerklePatriciaVerifier.sol`
- `Rlp.sol`
- `UniswapOracle.sol`
- `UQ112x112.sol`

Impact

Because correctness is so critically important when developing smart contracts, we recommend that all projects strive for 100% code coverage. Testing should be an essential part of the software development lifecycle. No matter how simple a function may be, untested code is always prone to bugs.

Recommendations

Expand the test suite so that all functions and their branches are covered by unit or integration tests.

Remediation

The issue has been acknowledged by Ionic Protocol and no changes have been made.

Ionic Protocol states that “The untested codebase is strictly related to the Keydonix

implementation, which we currently have no plans of using or deploying. In case we proceed with their usage, we'll proceed with a re-audit and development of the tests for those smart contracts. FusePoolLens.sol and FusePoolLensSecondary.sol are instead tested extensively at the integration level, being core components of both how our SDK fetches data from fuse pools, as well as how the UI displays such data."

4 Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment.

4.1 Strategy contracts

In addition to reviewing the 4626 vault strategies, we also checked the the external vaults with which the strategy contracts interact. It is important to validate that an attacker cannot manipulate a value in the external contracts that the strategies rely upon. For example, the function `totalAssets` is used to calculate the number of shares and assets based on data from external contracts. If an attacker is able to manipulate the `totalAssets` and call strategy functions before the state of the vault has been updated, then the number of shares issued to the attacker may be inflated.

For example, in the external contract `AlpacaVault`, the `work` function executes an external call before updating the `vaultDebtVal` while also transferring out tokens. This is used when calculating `totalToken`, which the `AlpacaERC4626` contract relies upon in the `totalAssets()` to calculate the total amount of underlying tokens the vault holds.

This can lead to a manipulation of `totalAssets` during a `work` call, inflating the shares minted. Currently, the use of a non-reentrant modifier for external functions, such as `deposit` and `withdraw` in `AlpacaVault`, called from `afterDeposit` and `beforeWithdraw` prevents the attack from executing successfully. For more information on similar attacks, read this and this.

```
function convertToShares(uint256 assets) public view returns (uint256) {
  uint256 supply = totalSupply; // Saves an extra SLOAD if totalSupply is
    non-zero.

  return supply == 0 ? assets : assets.mulDivDown(supply, totalAssets());
}
```

The share price is based on `totalAssets()`

```
function totalAssets() public view override returns (uint256) {
  return
    alpacaVault.balanceOf(address(this)).mulDivDown(alpacaVault.totalToken()
    alpacaVault.totalSupply());
```

```
}
```

That depends on `totalToken()`, which can be artificially deflated.

Developers should be aware that an attacker may be able to use an external call from the vault to make a reentrant call or a call to an attacker controlled contract so a vault's external calls should be scrutinized. This is very important when external data/calls are used to calculate share prices of an 4626 vault.

4.2 Re-entrancy in Compound

We were requested to provide a cursory review of re-entrancy issues in compound right after the rari hack. Following are our recommendations:

- Consider updating the `doTransferOut` in `CEther.sol` to prevent possible re-entrancy.

```
function doTransferOut(address to, uint256 amount) internal override {
  // Send the Ether and revert on failure
  (bool success, ) = to.call{ value: amount }("");
  to.transfer(amount);
  require(success, "doTransferOut failed");
}
```

- Consider updating the `borrowInternal` in `CToken.sol` to correctly follow check-effects-interaction, preventing exploitation from exotic ERC20s with callbacks.

```
function borrowFresh(address borrower, uint256 borrowAmount)
  internal returns (uint256) {
    ...
  doTransferOut(borrower, borrowAmount);

  /* We write the previously calculated values into storage */
  accountBorrows[borrower].principal = vars.accountBorrowsNew;
  accountBorrows[borrower].interestIndex = borrowIndex;
  totalBorrows = vars.totalBorrowsNew;

  doTransferOut(borrower, borrowAmount);
```

- If possible, merge patches from latest `Compound` codebase. This would prevent

any known issues from inadvertently causing unexpected problems in the future.

4.3 Documentation

As a consequence of updating the Fuse codebase, which is a modified fork of the Compound codebase, many aspects of the documentation are either outdated, inaccurate or insufficient. The codebase is more difficult to read and reason about than it should be due to the interchangeable use of `NativeToken` and `ETH`.

Code maturity is very important in high-assurance projects. Undocumented code may result in developer confusion, potentially leading to future bugs, should the code need to be modified later on. In general, lack of documentation impedes auditors and external developers from reading, understanding, and extending the code. The problem will also be carried over if the code is ever forked or re-used.