Tidal

Date	May 2023
Auditors	Heiko Fisch, David Oz

1 Executive Summary

This report presents the results of our engagement with **Tidal** to review a subset of the Tidal Contracts V2.

The review was conducted over one week, from **May 8–15, 2023**, by **David Oz** and **Heiko Fisch**. A total of 10 person-days were spent.

An additional week of review was conducted from **June 5th 2023** by **David Oz** to review the mitigations proposed by the **Tidal** team.

We identified 3 critical and 2 major issues as well as several issues of medium or minor severity. Our recommendation is to review all findings, implement fixes and improvements, and then have the codebase undergo internal reviews and finally another external audit. We also propose improving the test coverage, rigorous testing campaigns on testnet, and a soft launch before going to full production mode with large user deposits.

The Tidal team describes their system as "cross-chain insurance marketplace with enhanced capital efficiency." Very briefly, buyers can buy insurance for a fixed policy and period, while sellers deposit the necessary collateral to provide coverage. Collateral can be withdrawn with a time delay. There is a pool manager with some privileges, but a more important role is played by the committee: Proposals for claims or to change important system parameters can be made by the pool manager or a committee member, and within a fixed time interval, committee members can vote on the proposal. If a certain threshold of affirmative votes is reached, the proposal has passed and can be processed. *Any threshold-sized subset of the committee has complete control over the funds held by the contract*, and it is important to communicate this trust assumption to the community. Moreover, we recommend setting a sufficiently high threshold and following standard multisig best practices to protect against common risks like loss of private keys, malicious insiders, etc.

Another centralization risk lies in the upgradeability of the contracts. However, the proxy contract as well as the upgrade mechanism and privileges have not been in scope for this engagement.

2 Scope

Our review focused on the commit hash `741e920cb0ce9acb1d1aa4f1e2b6529ae274a4dd`. The list of files in scope can be found in the Appendix.

2.1 Objectives

Together with the **Tidal** team, we identified the following priorities for our review:

1. Correctness of the implementation, consistent with the intended functionality and without unintended edge cases.
2. Identify known vulnerabilities particular to smart contract systems, as outlined in our Smart Contract Best Practices, and the Smart Contract Weakness Classification Registry.

3 Findings

Each issue has an assigned severity:

- Minor issues are subjective in nature. They are typically suggestions around best practices or readability. Code maintainers should use their own judgment as to whether to address such issues.
- Medium issues are objective in nature but are not security vulnerabilities. These should be addressed unless there is a clear reason not to.
- Major issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- Critical issues are directly exploitable security vulnerabilities that need to be fixed.

3.1 `addPremium` – A Back Runner May Cause an Insurance Holder to Lose Their Refunds by Calling `addPremium` Right After the Original Call Critical ✓ Fixed

Resolution

Fixed in 3bbafab926df0ea39f444ef0fd5d2a6197f99a5d by returning with no-op in case `incomeMap[policyIndex_][week] = 0`, and by doing this eliminate the risk of loss of refunds.

Description

`addPremium` is a public function that can be called by anyone and that distributes the weekly premium payments to the pool manager and the rest of the pool share holders. If the collateral deposited is not enough to cover the total coverage offered to insurance holders for a given week, refunds are allocated pro rata for all insurance holders of that particular week and policy. However, in the current implementation, attackers can call `addPremium` right after the original call to `addPremium` but before the call to `refund`; this will cause the insurance holders to lose their refunds, which will be effectively locked forever in the contract (unless the contract is upgraded).

Examples

contracts/Pool.sol:L313-L314

```
refundMap[policyIndex_][week] = incomeMap[policyIndex_][week].mul(
    allCovered.sub(maximumToCover)).div(allCovered);
```

Recommendation

`addPremium` should contain a validation check in the beginning of the function that reverts for the case of `incomeMap[policyIndex_][week] = 0`.

3.2 `refund` – Attacker Can Lock Insurance Holder's Refunds by Calling `refund` Before a Refund Was Allocated Critical ✓ Fixed

Resolution

Fixed in 3bbafab926df0ea39f444ef0fd5d2a6197f99a5d by implementing the auditor's recommendation.

Description

`addPremium` is used to determine the refund amount that an insurance holder is eligible to claim. The amount is stored in the `refundMap` mapping and can then later be claimed by anyone on behalf of an insurance holder by calling `refund`. The `refund` function can't be called more than once for a given combination of `policyIndex_`, `week_`, and `who_`, as it would revert with an "Already refunded" error. This gives an attacker the opportunity to call `refund` on behalf of any insurance holder with value 0 inside the `refundMap`, causing any future refund allocated for that holder in a given week and for a given policy to be locked forever in the contract (unless the contract is upgraded).

Examples

contracts/Pool.sol:L341-L367

```
function refund(
    uint256 policyIndex_,
    uint256 week_,
    address who_
) external noReenter {
    Coverage storage coverage = coverageMap[policyIndex_][week_][who_];

    require(!coverage.refunded, "Already refunded");

    uint256 allCovered = coveredMap[policyIndex_][week_];
    uint256 amountToRefund = refundMap[policyIndex_][week_].mul(
        coverage.amount).div(allCovered);
    coverage.amount = coverage.amount.mul(
        coverage.premium.sub(amountToRefund)).div(coverage.premium);
    coverage.refunded = true;

    IERC20(baseToken).safeTransfer(who_, amountToRefund);

    if (eventAggregator != address(0)) {
        IEventAggregator(eventAggregator).refund(
            policyIndex_,
            week_,
            who_,
            amountToRefund
        );
    }
}
```

Recommendation

There should be a validation check at the beginning of the function that reverts if `refundMap[policyIndex_][week_] == 0`.

3.3 `addTidal`, `_updateUserTidal`, `withdrawTidal` – Wrong Arithmetic Calculations Critical ✓ Fixed

Resolution

Fixed in 3bbafab926df0ea39f444ef0fd5d2a6197f99a5d by implementing the auditor's recommendation.

Description

To further incentivize sellers, anyone – although it will usually be the pool manager – can send an arbitrary amount of the Tidal token to a pool, which is then supposed to be distributed proportionally among the share owners. There are several flaws in the calculations that implement this mechanism:

A. `addTidal` :

contracts/Pool.sol:L543-L544

```
poolInfo.accTidalPerShare = poolInfo.accTidalPerShare.add(
    amount_.mul(SHARE_UNITS)).div(poolInfo.totalShare);
```

This should be:

```
poolInfo.accTidalPerShare = poolInfo.accTidalPerShare.add(
    amount_.mul(SHARE_UNITS).div(poolInfo.totalShare));
```

Note the different parenthesization. Without `SafeMath` :

```
poolInfo.accTidalPerShare += amount_ * SHARE_UNITS / poolInfo.totalShare;
```

B. `_updateUserTidal` :

contracts/Pool.sol:L549-L550

```
uint256 accAmount = poolInfo.accTidalPerShare.add(
    userInfo.share).div(SHARE_UNITS);
```

This should be:

```
uint256 accAmount = poolInfo.accTidalPerShare.mul(
    userInfo.share).div(SHARE_UNITS);
```

Note that `add` has been replaced with `mul` . Without `SafeMath` :

```
uint256 accAmount = poolInfo.accTidalPerShare * userInfo.share / SHARE_UNITS;
```

C. `withdrawTidal` :

contracts/Pool.sol:L568

```
uint256 accAmount = poolInfo.accTidalPerShare.add(userInfo.share);
```

As in B, this should be:

```
uint256 accAmount = poolInfo.accTidalPerShare.mul(
    userInfo.share).div(SHARE_UNITS);
```

Note that `add` has been replaced with `mul` and that a division by `SHARE_UNITS` has been appended. Without `SafeMath` :

```
uint256 accAmount = poolInfo.accTidalPerShare * userInfo.share / SHARE_UNITS;
```

As an additional minor point, the division in `addTidal` will revert with a panic (0x12) if the number of shares in the pool is zero. This case could be handled more gracefully.

Recommendation

Implement the fixes described above. The versions without `SafeMath` are easier to read and should be preferred; see issue 3.13.

3.4 `claim` – Incomplete and Lenient Implementation Major ✓ Fixed

Resolution

Acknowledged but not fixed in this version. The client provided the following message: "No fix for this version. This one is not a bug in the code but is a missing feature on product logic. The product is good for release without a fix. We may implement related functions in the future."

Description

In the current version of the code, the `claim` function is lacking crucial input validation logic as well as required state changes. Most of the process is implemented in other contracts or off-chain at the moment and is therefore out of scope for this audit, but there might still be issues caused by potential errors in the process. Moreover, pool manager and committee together have unlimited ownership of the deposits and can essentially withdraw all collateral to any desired address.

Examples

contracts/Pool.sol:L588-L592

```
function claim(
    uint256 policyIndex_,
    uint256 amount_,
    address receipient_
) external onlyPoolManager {
```

Recommendation

To ensure a more secure claiming process, we propose adding the following logic to the `claim` function:

1. `refund` should be called at the beginning of the `claim` flow, so that the recipient's true coverage amount will be used.
2. `policyIndex` should be added as a parameter to this function, so that `coverageMap` can be used to validate that the amount claimed on behalf of a recipient is covered.
3. The payout amount should be subtracted in the `coveredMap` and `coverageMap` mappings.

3.5 `buy` – Insurance Buyers Trying to Increase Their Coverage Amount Will Lose Their Previous Coverage Major ✓ Fixed

Resolution

Fixed in 3bbafab926df0ea39f444ef0fd5d2a6197f99a5d by implementing the auditor's recommendation.

Description

When a user is willing to buy insurance, he is required to specify the desired amount (denoted as `amount_`) and to pay the entire premium upfront. In return, he receives the ownership over an entry inside the `coverageMap` mapping. If a user calls the `buy` function more than once for the same policy and time frame, his entry in the `coverageMap` will not represent the *accumulated* amount that he paid for but only the *last* coverage amount, which means previous coverage will be lost forever (unless the contract is upgraded).

Examples

contracts/Pool.sol:L266-L280

```
for (uint256 w = fromWeek_; w < toWeek_; ++w) {
    incomeMap[policyIndex_][w] =
        incomeMap[policyIndex_][w].add(premium);
    coveredMap[policyIndex_][w] =
        coveredMap[policyIndex_][w].add(amount_);

    require(coveredMap[policyIndex_][w] <= maximumToCover,
        "Not enough to buy");

    coverageMap[policyIndex_][w][_msgSender()] = Coverage({
        amount: amount_,
        premium: premium,
        refunded: false
    });
}
```

Recommendation

The coverage entry that represents the user's coverage should not be overwritten but should hold the *accumulated* amount of coverage instead.

3.6 Several Issues Related to Upgradeability of Contracts Medium ✓ Fixed

Resolution

Partially fixed in 3bbafab926df0ea39f444ef0fd5d2a6197f99a5d as auditor's recommendations were implemented except from introducing `__gap` variables for `NonReentrancy` and `EventAggregator`.

Description

We did not find a proxy contract or factory in the repository, but the README contains the following information:

README.md:L11

```
Every Pool is a standalone smart contract. It is made upgradeable with OpenZeppelin's Proxy Upgrade Pattern.
```

README.md:L56

```
And there will be multiple proxies and one implementation of the Pools, and one proxy and one implementation of EventAggregator.
```

There are several issues related to upgradeability or, generally, using the contracts as implementations for proxies. All recommendations in this report assume that it is not necessary to remain compatible with an existing deployment.

A. The `Pool.sol` file imports `Initializable.sol` from OpenZeppelin's `contracts-upgradeable` and several other files from their "regular" `contracts` package.

contracts/Pool.sol:L5-L10

```
import "@openzeppelin/contracts-upgradeable/proxy/utils/Initializable.sol";

import "@openzeppelin/contracts/utils/Context.sol";
import "@openzeppelin/contracts/utils/math/SafeMath.sol";
import "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol";
```

These two should not be mixed, and in an upgradeable context, all files should be imported from `contracts-upgradeable`. Note that the import of `Ownable.sol` in `NonReentrancy.sol` can be removed completely; see issue 3.20.

B. If upgradeability is supposed to work with inheritance, there should be dummy variables at the end of each contract in the inheritance hierarchy. Some of these have to be removed when "real" state variables are added. More precisely, it is conventional to use a fixed-size `uint256` array `__gap`, such that the consecutively occupied slots at the beginning (for the "real" state variables) add up to 50 with the size of the array. If state variables are added later, the gap's size has to be reduced accordingly to maintain this invariant. Currently, the contracts do not declare such a `__gap` variable.

C. Implementation contracts should not remain uninitalized. To prevent initialization by an attacker – which, in some cases, can have an impact on the proxy – the implementation contract's constructor should call `_disableInitializers`.

Recommendation

1. Refamiliarize yourself with the subtleties and pitfalls of upgradeable contracts, in particular regarding state variables and the storage gap. A lot of useful information can be found here.
2. Only import from `contracts-upgradeable`, not from `contracts`.
3. Add appropriately-sized storage gaps at least to `PoolModel`, `NonReentrancy`, and `EventAggregator`. (Note that adding a storage gap to `NonReentrancy` will break compatibility with existing deployments.) Ideally, add comments and warnings to each file that state variables may only be added at the end, that the storage gap's size has to be reduced accordingly, and that state variables must not be removed, rearranged, or in any way altered (e.g., type, `constant`, `immutable`). No state variables should ever be added to the `Pool` contract, and a comment should make that clear.
4. Add a constructor to `Pool` and `EventAggregator` that calls `_disableInitializers`.

3.7 `initialize` – Committee Members Array Can Contain Duplicates Medium ✓ Fixed

Resolution

Fixed in 3bbafab926df0ea39f444ef0fd5d2a6197f99a5d by implementing a nested loop to de-duplicate committee members.

Description

The initial committee members are given as array argument to the pool's `initialize` function. When the array is processed, there is no check for duplicates, and duplicates may also end up in the storage array `committeeArray`.

contracts/Pool.sol:L43-L47

```
for (uint256 i = 0; i < committeeMembers_.length; ++i) {
    address member = committeeMembers_[i];
    committeeArray.push(member);
    committeeIndexPlusOne[member] = committeeArray.length;
}
```

Duplicates will result in a discrepancy between the length of the array – which is later interpreted as the number of committee members – and the actual number of (different) committee members. This could lead to more problems, such as an insufficient committee size to reach the threshold.

Recommendation

The `initialize` function should verify in the loop that `member` hasn't been added before. Note that `_executeAddToCommittee` refuses to add someone who is already in the committee, and the same technique can be employed here.

3.8 `addPolicy`, `setPolicy` – Missing Input Validation Medium ✓ Fixed

Resolution

Fixed in 3bbafab926df0ea39f444ef0fd5d2a6197f99a5d by implementing the auditor's recommendation.

Description and Recommendation

Both `addPolicy` and `setPolicy` are missing essential input validation on two main parameters:

1. `collateralRatio_` – Should be validated to be non-zero, and it might be worth adding a range check.
2. `weeklyPremium_` – Should be less than `RATIO_BASE` at least, and it might be worth adding a maximum value check.

Examples

contracts/Pool.sol:L159

```
function addPolicy(
```

contracts/Pool.sol:L143

```
function setPolicy(
```

3.9 `Pool.buy` – Users May End Up Paying More Than Intended Due to Changes in `policy.weeklyPremium` Medium ✓ Fixed

Resolution
Fixed in 3bbafab926df0ea39f444ef0fd5d2a6197f99a5d by implementing the auditor's recommendation.

Description

The price that an insurance buyer has to pay for insurance is determined by the duration of the coverage and the `weeklyPremium`. The price increases as the `weeklyPremium` increases. If a `buy` transaction is waiting in the mempool but eventually front-run by another transaction that increases `weeklyPremium`, the user will end up paying more than they anticipated for the same insurance coverage (assuming their allowance to the `Pool` contract is unlimited or at least higher than what they expected to pay).

Examples

contracts/Pool.sol:L273-L274

```
uint256 premium = amount_.mul(policy.weeklyPremium).div(RATIO_BASE);
uint256 allPremium = premium.mul(toWeek_.sub(fromWeek_));
```

Recommendation

Consider adding a parameter for the maximum amount to pay, and make sure that the transaction will revert if `allPremium` is greater than this maximum value.

3.10 Missing Validation Checks in `execute` Medium ✓ Fixed

Resolution
Fixed in 3bbafab926df0ea39f444ef0fd5d2a6197f99a5d by implementing the auditor's recommendations.

Description

The `Pool` contract implements a threshold voting mechanism for some changes in the contract state, where either the pool manager or a committee member can propose a change by calling `claim`, `changePoolManager`, `addToCommittee`, `removeFromCommittee`, or `changeCommitteeThreshold`, and then the committee has a time period for voting. If the threshold is reached during this period, then anyone can call `execute` to execute the state change.

While some validation checks are implemented in the proposal phase, this is not enough to ensure that business logic rules around these changes are completely enforced.

1. `_executeRemoveFromCommittee` – While the `removeFromCommittee` function makes sure that `committeeArray.length > committeeThreshold`, i.e., that there should always be enough committee members to reach the threshold, the same validation check is not enforced in `_executeRemoveFromCommittee`. To better illustrate the issue, let's consider the following example: `committeeArray.length = 5`, `committeeThreshold = 4`, and now `removeFromCommittee` is called two times in a row, where the second call is made before the first call reaches the threshold. In this case, both requests will be executed successfully, and we end up with `committeeArray.length = 3` and `committeeThreshold = 4`, which is clearly not desired.
2. `_executeChangeCommitteeThreshold` – Applying the same concept here, this function lacks the validation check of `threshold_ <= committeeArray.length`, leading to the same issue as above. Let's consider the following example: `committeeArray.length = 3`, `committeeThreshold = 2`, and now `changeCommitteeThreshold` is called with `threshold_ = 3`, but before this request is executed, `removeFromCommittee` is called. After both requests have been executed successfully, we will end up with `committeeThreshold = 3` and `committeeArray.length = 2`, which is clearly not desired.

Examples

contracts/Pool.sol:L783

```
function _executeRemoveFromCommittee(address who_) private {
```

contracts/Pool.sol:L796

```
function _executeChangeCommitteeThreshold(uint256 threshold_) private {
```

Recommendation

Apply the same validation checks in the functions that execute the state change.

3.11 Reentrancy Concerns Minor ✓ Fixed

Resolution

Fixed in 3bbafab926df0ea39f444ef0fd5d2a6197f99a5d by implementing the auditor's recommendations.

Description and Recommendation

A. All external functions in the `Pool` contract that make calls to the base or the Tidal token – and only these – have a `noReenter` modifier. That means that it is not possible to reenter the contract *through these functions*, but it could still be possible to reenter the pool through a *different* external or public function that does not have such a modifier. Assuming the token contract allows reentrancy, the following could happen, for instance:

1. Alice calls `withdrawReady` .
2. During the call to the token contract, Alice gets control of execution through a callback.
3. She reenters the pool contract through the `withdraw` function.

Note that, at this point, `userInfo.pendingWithdrawShare` has a "wrong" value because we left the `Pool` contract before this state variable was updated. So the reentering call is operating on inconsistent state.

We didn't find a way to cause actual harm through this or similar reentrancies, but to rely on this kind of reasoning is dangerous, and there's always the risk to miss something. It is, therefore, recommended to add a `noReenter` modifier to all state-changing external functions, in particular the ones operating with shares.

B. A second concern is reentrancy through `view` functions. In the example above, note that when we leave the pool contract, it is not only `userInfo.pendingWithdrawShare` that hasn't been updated yet, it is also `poolInfo.pendingWithdrawShare` . Hence, if we call, for example, `getAvailableCapacity` in step number 3, we will get a wrong result.

If this or other `view` functions are supposed to give reliable results under all circumstances, they should revert if `islocked` is `true` . (This state variable is currently private and not accessible in the derived contract `Pool` , so a small change has to be made in the `NonReentrancy` contract, too.)

3.12 Hard-Coded Minimum Deposit Amount Minor ✓ Fixed

Resolution

Fixed in 3bbafab926df0ea39f444ef0fd5d2a6197f99a5d by implementing the auditor's recommendation.

Description

The `deposit` function specifies a minimum amount of 1e12 units of the base token for a deposit:

contracts/Pool.sol:L22

```
uint256 constant AMOUNT_PER_SHARE = 1e18;
```

contracts/Pool.sol:L369-L376

```
// Anyone can be a seller, and deposit baseToken (e.g. USDC or WETH)
// to the pool.
function deposit(
    uint256 amount_
) external noReenter {
    require(enabled, "Not enabled");

    require(amount_ >= AMOUNT_PER_SHARE / 1000000, "Less than minimum");
```

Whether that's an appropriate minimum amount or not depends on the base token. Note that the two example tokens listed above are USDC and WETH. With current ETH prices, 1e12 Wei cost an affordable 0.2 US Cent. USDC, on the other hand, has 6 decimals, so 1e12 units are worth 1 million USD, which is … steep.

Recommendation

The minimum deposit amount should be configurable.

3.13 Unnecessary Use of `SafeMath` Library Minor ✓ Fixed

Resolution

Fixed in 3bbafab926df0ea39f444ef0fd5d2a6197f99a5d by implementing the auditor's recommendation.

Description

Since Solidity v0.8.0, all arithmetic operations are checked by default and revert on over- or underflow. Hence, it is not necessary anymore to use the `SafeMath` library (or `SafeMathUpgradeable`). Employing it nonetheless not only wastes gas but also reduces the readability of arithmetic expressions considerably.

Examples

The assignment

```
poolInfo.accTidalPerShare = poolInfo.accTidalPerShare.add(amount_.mul(SHARE_UNITS).div(poolInfo.totalShare));
```

is a lot easier to read without `SafeMath`:

```
poolInfo.accTidalPerShare += amount_ * SHARE_UNITS / poolInfo.totalShare;
```

See also issue 3.3.

Recommendation

We recommend using the built-in arithmetic operations instead of `SafeMath`.

3.14 Outdated Solidity Version Minor ✓ Fixed

Resolution

Fixed in 3bbafab926df0ea39f444ef0fd5d2a6197f99a5d by implementing the auditor's recommendation.

Description

The source files' version pragmas either specify that they need compiler version exactly 0.8.10 or at least 0.8.10:

contracts/Pool.sol:L2

```
pragma solidity 0.8.10;
```

contracts/helper/EventAggregator.sol:L2

```
pragma solidity ^0.8.10;
```

Solidity v0.8.10 is a fairly dated version that has known security issues. We generally recommend using the latest version of the compiler (at the time of writing, this is v0.8.20), and we also discourage the use of floating pragmas to make sure that the source files are actually compiled and deployed with the same compiler version they have been tested with.

Recommendation

Use the Solidity compiler v0.8.20, and change the version pragma in all Solidity source files to `pragma solidity 0.8.20;`.

3.15 Code Used for Testing Purposes Should Be Removed Before Deployment Minor ✓ Fixed

Resolution

Fixed in 49d6afd6abc463dd3fde3df0df715c475bb3e013 by implementing the auditor's recommendation.

Description

Variables and logic have been added to the code whose only purpose is to make it easier to test. This might cause unexpected behavior if deployed in production. For instance, `onlyTest` and `setTimeExtra` should be removed from the code before deployment, as well as `timeExtra` in `getCurrentWeek` and `getNow`.

Examples

contracts/Pool.sol:L55

```
modifier onlyTest() {
```

contracts/Pool.sol:L67

```
function setTimeExtra(uint256 timeExtra_) external onlyTest {
```

contracts/Pool.sol:L71-L73

```
function getCurrentWeek() public view returns(uint256) {
    return (block.timestamp + TIME_OFFSET + timeExtra) / (7 days);
}
```

contracts/Pool.sol:L75-L77

```
function getNow() public view returns(uint256) {
    return block.timestamp + timeExtra;
}
```

Recommendation

For the long term, consider mimicking this behavior by using features offered by your testing framework.

3.16 Missing Events Minor ✓ Fixed

Resolution

Fixed in 3bbafab926df0ea39f444ef0fd5d2a6197f99a5d by implementing the auditor's recommendations.

Description

Some state-changing functions do not emit an event at all or omit relevant information.

Examples

A. `Pool.setEventAggregator` should emit an event with the value of `eventAggregator_` so that off-chain services will be notified and can automatically adjust.

contracts/Pool.sol:L93-L95

```
function setEventAggregator(address eventAggregator_) external onlyPoolManager {
    eventAggregator = eventAggregator_;
}
```

B. `Pool.enablePool` should emit an event when the pool is dis- or enabled.

contracts/Pool.sol:L581-L583

```
function enablePool(bool enabled_) external onlyPoolManager {
    enabled = enabled_;
}
```

C. `Pool.execute` only logs the `requestIndex_` while it should also include the `operation` and `data` to better reflect the state change in the transaction.

contracts/Pool.sol:L756-L760

```
if (eventAggregator != address(0)) {
    IEventAggregator(eventAggregator).execute(
        requestIndex_
    );
}
```

Recommendation

State-changing functions should emit an event to have an audit trail and enable monitoring of smart contract usage.

3.17 `CommitteeRequest`, `WithdrawRequest` - Should Use an `enum` Type ✓ Fixed

Resolution

Fixed in 3bbafab926df0ea39f444ef0fd5d2a6197f99a5d by implementing the auditor's recommendation.

Description and Recommendation

A. There are 5 different operations: `claim`, `changePoolManager`, `addToCommittee`, `removeFromCommittee`, and `changeCommitteeThreshold`. These operations are numbered from 0 to 4, and this number is stored as a `uint8` in committee requests:

contracts/model/PoolModel.sol:L99-L104

```
struct CommitteeRequest {
    uint256 time;
    uint256 vote;
    bool executed;
    uint8 operation;
    bytes data;
```

Developers have to remember or look up which number denotes which operation:

contracts/Pool.sol:L738-L754

```
if (cr.operation == 0) {
    (uint256 amount, address receipient) = abi.decode(
        cr.data, (uint256, address));
    _executeClaim(amount, receipient);
} else if (cr.operation == 1) {
    address poolManager = abi.decode(cr.data, (address));
    _executeChangePoolManager(poolManager);
} else if (cr.operation == 2) {
    address newMember = abi.decode(cr.data, (address));
    _executeAddToCommittee(newMember);
} else if (cr.operation == 3) {
    address oldMember = abi.decode(cr.data, (address));
    _executeRemoveFromCommittee(oldMember);
} else if (cr.operation == 4) {
    uint256 threshold = abi.decode(cr.data, (uint256));
    _executeChangeCommitteeThreshold(threshold);
}
```

This is error-prone and tedious. An enum type is a safer and more convenient way to encode the different operations. In fact, this is a textbook scenario for employing an enum, and we recommend doing so.

B. Withdrawal requests are first created with the `withdraw` function. After `withdrawWaitWeeks1` weeks, they can be advanced to a "pending" status by calling `withdrawPending`. Finally, after another `withdrawWaitWeeks2` weeks, the request can be executed via `withdrawReady`.

This is currently implemented via two boolean members in the `WithdrawRequest` struct, `pending` and `executed`:

contracts/model/PoolModel.sol:L72-L78

```
struct WithdrawRequest {
    uint256 share;
    uint256 time;
    bool pending;
    bool executed;
    bool succeeded;
}
```

Initially, when the request is created, they're both set to `false`. For a pending request, `pending` is `true`, and `executed` remains at `false`. Finally, they're both set to `true` for an executed request.

An object transitioning through a series of states is another excellent use case for enums. In this example, the state could be modeled with an enum as follows: `enum Status { Created, Pending, Executed }`. This approach has several advantages compared to the implementation with two boolean variables:

- It uses only one variable, instead of two. In particular, setting and querying the state only involves one variable.
- It can be easily extended to more states without introducing additional variables.
- The object can never be in more than one state at once or in an undefined state. (With the current implementation, it would be possible to have `pending == false` and `executed == true`.)

Remark

It is often a good idea to have something like "None" or "NonExistent" as first value in the enum. That makes it easy to distinguish "real" objects from unchanged storage, as in: "Here is no object." In the two examples above, that is not necessary, but it wouldn't hurt either.

3.18 No NatSpec Annotations

Description

NatSpec is the de facto standard for the annotation of Solidity files. To quote the Solidity documentation:

> It is recommended that Solidity contracts are fully annotated using NatSpec for all public interfaces (everything in the ABI).

The Tidal codebase does not use NatSpec, and there's not a lot of documentation and comments in general.

Recommendation

Use NatSpec documentation and follow the advice in the quote.

3.19 `vote` - Voting "No" Has No Effect ✓ Fixed

Resolution

Fixed in 3bbafab926df0ea39f444ef0fd5d2a6197f99a5d by implementing the auditor's recommendation.

Description

Committee members can vote on proposals with either "yes" or "no". Voting "no" has no effect at all, i.e., there is no state change or event emitted, no return value, etc.

contracts/Pool.sol:L695-L701

```
function vote(
    uint256 requestIndex_,
    bool support_
) external onlyCommittee {
    if (!support_) {
        return;
    }
```

This means voting with "no" is pointless, and the option to do so could be removed completely.

Recommendation

Consider removing the `bool support_` parameter from the `vote` function, such that calling `vote` is always a "yes" vote. Maybe rename the function to make this more explicit.

3.20 Unused Import ✓ Fixed

Resolution

Fixed in 3bbafab926df0ea39f444ef0fd5d2a6197f99a5d by implementing the auditor’s recommendation.

Description

The file `NonReentrancy.sol` imports `Ownable.sol`, but this import is not used.

contracts/common/NonReentrancy.sol:L4

```
import "@openzeppelin/contracts/access/Ownable.sol";
```

Recommendation

Remove the unnecessary import.

3.21 Unnecessary and Outdated Pragma Directive ✓ Fixed

Resolution
Fixed in 3bbafab926df0ea39f444ef0fd5d2a6197f99a5d by implementing the auditor’s recommendation.

Description

The `Pool.sol` source file uses the pragma directive `pragma experimental ABIEncoderV2;`:

contracts/Pool.sol:L3

```
pragma experimental ABIEncoderV2;
```

ABI coder V2 is the default since Solidity v0.8.0 and is considered non-experimental as of Solidity v0.6.0. Hence, this directive is not necessary and even a bit misleading because the “experimental” status was removed long ago.

Recommendation

This line can be removed. If you want to be explicit for some reason, it should be replaced with `pragma abicoder v2;`.

3.22 `vote` Could Call `execute` When `committeeThreshold` Is Reached ✓ Fixed

Resolution
Fixed in 3bbafab926df0ea39f444ef0fd5d2a6197f99a5d by implementing the auditor’s recommendation.

Description and Recommendation

In the current version of the code, an additional transaction to `execute` is needed in case the threshold was reached for a specific request. Instead, `execute` could be invoked as part of `vote` when the threshold is reached.

contracts/Pool.sol:L714

```
cr.vote = cr.vote.add(1);
```

Appendix 1 - Files in Scope

This audit covered the following files:

File Name	SHA-1 Hash
contracts/Pool.sol	bde682116b477e2a7ddbc797fefaa0dcd76ace20
contracts/model/PoolModel.sol	bb4dfc828e9c4b1bbeafe13c25a87403d6c33c0a
contracts/interface/IEventAggregator.sol	6c337d6598398e01a7a9afc98fea96e83e80456b
contracts/helper/EventAggregator.sol	e1fa13dc00a8bcfdbbf9a8b952259dfa1cd63be3
contracts/common/NonReentrancy.sol	8ae831e28d8873a41bd3d9a18f8f637be8033318

Appendix 2 - Disclosure