# GoodGhosting V2 Pool

Smart Contract Security Assessment

June 16, 2022

## ABSTRACT

Dedaub was commissioned to perform a security audit of the GoodGhosting V2 Pool protocol.

The first version of the protocol was also audited by us in the past. This is a full audit of the V2 of the protocol from the goodghosting-protocol-v2 repository, up to commit 83d37b9b82c46e8bcdae724f89bd5daa7b08dcef. Resolved items are per commit 4f535bca3b47657ae49b82a8f9f1606d2bf820ff, of July 20, 2022.

## Setting and Caveats

The audit's main target is security threats, i.e., what the community understanding would likely call "hacking", rather than regular use of the protocol. Functional correctness (i.e., issues in "regular use") is a secondary consideration. Typically it can only be covered if we are provided with unambiguous (i.e., full-detail) specifications of what is the expected, correct behavior. In terms of functional correctness, we often trusted the code's calculations and interactions, in the absence of any other specification. Functional correctness relative to low-level calculations (including units, scaling, quantities returned from external protocols) is generally most effectively done through thorough testing rather than human auditing. The scope of the audit includes smart contract code. Interactions with off-chain (front-end or back-end) code are not examined other than to consider entry points for the contracts, i.e., calls into a smart contract that may disrupt the contract's functioning.

The resolution of report items is determined by local inspection of changes, not a full re-audit. Since there was a large volume of changes and significant time elapsed, the development team is advised to be especially vigilant with testing the consequences of fixes performed after the initial audit.

## VULNERABILITIES & FUNCTIONAL ISSUES

This section details issues that affect the functionality of the contract. Dedaub generally categorizes issues according to the following severities, but may also take other considerations into account such as impact or difficulty in exploitation:

| Category | Description |
|---|---|
| CRITICAL | Can be profitably exploited by any knowledgeable third party attacker to drain a portion of the system's or users' funds OR the contract does not function as intended and severe loss of funds may result. |
| HIGH | Third party attackers or faulty functionality may block the system or cause the system or users to lose funds. Important system invariants can be violated. |
| MEDIUM | Examples:<br>-User or system funds can be lost when third party systems misbehave.<br>-DoS, under specific conditions.<br>-Part of the functionality becomes unusable due to programming error. |
| LOW | Examples:<br>-Breaking important system invariants, but without apparent consequences.<br>-Buggy functionality for trusted users where a workaround exists.<br>-Security issues which may manifest when the system evolves. |

Issue resolution includes "dismissed", by the client, or "resolved", per the auditors.

## CRITICAL SEVERITY:

| ID | Description | STATUS |
|---|---|---|
| C1 | `redeemFromExternalPoolForFixedDepositPool` can be called for flexible pools | **RESOLVED** |

`redeemFromExternalPoolForFixedDepositPool` is called automatically during the first `withdraw()`, and only for fixed deposit pools.

```
if (!flexibleSegmentPayment) {
        // First player to withdraw redeems everyone's funds
        if (!redeemed) {
            redeemFromExternalPoolForFixedDepositPool(_minAmount);
        }
} else {
        _setGlobalPoolParamsForFlexibleDepositPool();
}
```

However, this function can be also called directly by anyone once the game is completed, and a direct call could also happen for a **flexible** pool (there is no check that the pool is indeed flexible). Depending on the strategy, this could potentially leave the contract in an invalid state (with the possibility of a loss of funds).

- A call on a flexible pool would call `strategy.redeem` with `amount = 0`. This could in principle succeed for some strategies since we're asking for no tokens at all.

```
strategy.redeem(inboundToken, 0, flexibleSegmentPayment, _minAmount,
                disableRewardTokenClaim);
```

- `calculateAndUpdateGameAccounting` would be called with `_totalBalance = 0`, and `_calculateAndSetAdminAccounting` would be called with `grossInterest = 0`, leaving the contract in an invalid state (invalid values of `impermanentLossShare` / netTotalGamePrincipal / totalGameInterest / etc).

```
uint256 grossInterest =
  _calculateAndUpdateGameAccounting(totalBalance, grossRewardTokenAmount);
// shifting this after the game accounting since we need to emit a
// accounting event
if (redeemed) {
      revert FUNDS_REDEEMED_FROM_EXTERNAL_POOL();
}
redeemed = true;

_calculateAndSetAdminAccounting(grossInterest, grossRewardTokenAmount);
```

- `redeemed = true` would be set, which will impact future calls to `_setGlobalPoolParamsForFlexibleDepositPool`.

In any case, such problems can be easily fixed by checking that the pool is flexible.

| C2 | Possible loss of funds, by allowing swapping the funds of other players over a tilted AMM pool | **RESOLVED** |
|---|---|---|

A user can cause the protocol to swap a large sum of tokens **not owned by him** in an AMM. In such situations, "sandwich" attacks become possible, especially if slippage is controlled by the user. In a typical such attack:

1. A malicious user first buys himself a large quantity of the coin that the protocol tries to buy, hence tilting the pool causing the token's price to increase.
2. Then he forces the protocol to perform the swap, buying the token at a very high price (especially if a slippage bound is not set).
3. Finally the adversary sells the tokens he bought for a profit (since their price is raised).

GoodGhosting allows forcing a swap in two ways:

- First, any external user can force the protocol to redeem from an AMM strategy by calling `redeemFromExternalPoolForFixedDepositPool`. Note that `_minAmount` is controlled by the caller, so it can be set to 0.

```
function redeemFromExternalPoolForFixedDepositPool(uint256 _minAmount)
```

```
public virtual whenGameIsCompleted nonReentrant {
 uint256 totalBalance = 0;

 // Withdraws funds (principal + interest + rewards) from external pool
 strategy.redeem(inboundToken, 0, flexibleSegmentPayment, _minAmount,
                 disableRewardTokenClaim);
```

- Second, a participant can force the protocol to redeem the funds of all users of a fixed deposit pool, by being the first to call Pool::`withdraw`. Again, `_minAmount` is controlled by the caller.
  In this case, since the user participates in the game, he would also incur the loss of such an attack, so it might appear that he has no incentives to do so. Note, however, that the user causes the funds of **all users** to be redeemed. His own part could be very small, and the gain from the attack could substantially outweigh his loss.

```
function withdraw(uint256 _minAmount) external virtual {
      ...
  if (!flexibleSegmentPayment) {
      // First player to withdraw redeems everyone's funds
      if (!redeemed) {
            redeemFromExternalPoolForFixedDepositPool(_minAmount);
      }
  } else {
      _setGlobalPoolParamsForFlexibleDepositPool();
  }
  …
```

Note that these actions can be also performed by a malicious contract (instead of an EOA). This makes the attack particularly easy because all 3 attack steps can be performed in a single transaction. This avoids all risks involved with tilting the pool, and even allows the use of Flash Loans, which makes it possible to tilt large pools with limited funds.

To avoid such attacks, the protocol needs to ensure that:

- External users cannot force a redeem.

- Participants can only redeem their own funds, so any loss does not affect the funds of other users.

For instance, in flexible pools each user redeems his own funds. The same redeem logic could be employed for fixed deposit pools to avoid this issue.

| C3 | Possible loss of funds when players `earlyWithdraw` on a segment they haven't deposited in | **RESOLVED** |
|---|---|---|

The protocol follows a logic where, when a player early-withdraws, the contract subtracts the player's indexes from the cumulative index sum of the current segment.

This assumption can be valid only when the player that `earlyWithdraws` has made a deposit on this segment.

However, nothing prevents the player from waiting for the segment to change and withdraw then without depositing first. By doing so, he can cause a severe loss of funds for the rest of the players that remain in-game.

This stems from the following code segment:

`Pool:earlyWithdraw`

```
uint256 cumulativePlayerIndexSumForCurrentSegment =
    cumulativePlayerIndexSum[currentSegment];
for (uint256 i = 0; i <= players[msg.sender].mostRecentSegmentPaid; i++) {
    if (cumulativePlayerIndexSumForCurrentSegment != 0) {
        cumulativePlayerIndexSumForCurrentSegment =
           cumulativePlayerIndexSumForCurrentSegment.sub(
                    playerIndex[msg.sender][i]
           );
    } else {
        cumulativePlayerIndexSum[currentSegment - 1] =
           cumulativePlayerIndexSum[currentSegment - 1].sub(
                    playerIndex[msg.sender][i]
           );
    }
}
cumulativePlayerIndexSum[currentSegment] =
```

```
    cumulativePlayerIndexSumForCurrentSegment;
```

`cumulativePlayerIndexSumForCurrentSegment` equals 0 only when no one has deposited on the current segment, since for every new segment the cumulative index sum starts from 0 and the previous player indexes are being added upon depositing.

`Pool:_transferInboundTokenToContract`

```
uint256 cummalativePlayerIndexSumInMemory =
    cumulativePlayerIndexSum[currentSegment];
for (uint256 i = 0; i <= players[msg.sender].mostRecentSegmentPaid; i++) {
    cummalativePlayerIndexSumInMemory =
       cummalativePlayerIndexSumInMemory.add(playerIndex[msg.sender][i]);
}
cumulativePlayerIndexSum[currentSegment] = cummalativePlayerIndexSumInMemory;
```

Hence, if other players have made deposits and then a player withdraws without depositing, his indexes will be subtracted from the cumulative index sum of the remaining players.

There is another scenario where a player can fool the contract by joining the game with two different accounts and leverage this issue to steal all rewards from other players.

More specifically, the attacker-player can use one of the accounts to actually play in the game and try to keep a competitive position against other players and the other account to just perform the `earlyWithdraw` at the right time.

The goal is to drop the `cumulativePlayerIndexSum` below the second account index which will give him a >100% `playerSharePercentage`.

After the game ends, he needs to withdraw first to get all the rewards from all players since he will have a percentage greater than 100% along with any other player that have greater indexes than the “new” `cumulativePlayerIndexSum`.

The attacker-player doesn't even need to keep the two accounts in a condition where the secondary account will be entitled to exactly 100% of the profits–any greater number will be adjusted down to the full balance:

Pool:withdraw

```
if (payout > address(this).balance) {
    payout = address(this).balance;
}
```

## HIGH SEVERITY:

| ID | Description | STATUS |
|---|---|---|
| H1 | The netDepositAmount for a player can be inflated by tilting an external DEX pool | **RESOLVED** |

For AMM strategies (e.g., Curve), the code includes provisions of computing a "net" deposit amount, to counter slippage. However, this amount could be made *higher* than the actual deposit amount, by manipulating the AMM pool, thus giving the player an advantage.

Specifically, the code performs (`Pool::makeDeposit`):

```
uint256 netAmount = strategy.getNetDepositAmount(amount);
...
_transferInboundTokenToContract(_minAmount, amount, netAmount);
```

The returned amount of `strategy.getNetDepositAmount` can be influenced by tilting the AMM behind the strategy–the code merely computes the value of withdrawing a deposited token. E.g., for Mobius the code is:

```
function getNetDepositAmount(uint256 _amount) external view override
returns (uint256) {
```

```
        uint256[] memory amounts = new uint256[](2);
        amounts[0] = _amount;
        uint256 poolWithdrawAmount = pool.calculateTokenAmount(address(this),
                                                               amounts, true);
        return pool.calculateRemoveLiquidityOneToken(address(this),
                                                     poolWithdrawAmount, 0);
}
```

The subsequent call to `_transferInboundTokenToContract` just credits the user with the net deposit amount, while actually transferring the plain amount. The net deposit amount determines the player's index, i.e., their share of the pool:

```
uint256 currentSegmentplayerIndex =
      _netDepositAmount.mul(MULTIPLIER).div(block.timestamp);
playerIndex[msg.sender][currentSegment] = currentSegmentplayerIndex;
```

The intent behind the code is that the net deposit amount is lower than the actual deposited amount. But this is not ensured, and AMM pool manipulation can indeed make it higher. Whether such manipulation is profitable depends on the sizes of pools and game amounts. However, preventing this instance of the attack is easy, by ensuring that the net amount is lower than the deposited amount.

| H2 | Scaling deposits by block.timestamp does not achieve the intended effect | **RESOLVED** |
|---|---|---|

The protocol's documentation describes how investors that deposit earlier in a segment get a higher share of the profits. However, the code does not achieve this. The calculation in the code (in `_transferInboundTokenToContract`) is:

```
uint256 currentSegmentplayerIndex =
      _netDepositAmount.mul(MULTIPLIER).div(block.timestamp);
```

The unscaled use of `block.timestamp` (whose values are in the billions, so that a few minutes of difference have near-zero effect) means that every depositor practically gets the same index assigned.

| H3 | Users have incentives to withdraw last for gaining more rewards due to precision errors | **RESOLVED** |
|---|---|---|

The protocol updates the global parameters (including `cumulativePlayerIndexSum`) after each withdrawal for Flexible Pools.

```
function withdraw(uint256 _minAmount) external virtual {
    if (flexibleSegmentPayment) {
        totalGameInterest = totalGameInterest.sub(playerInterestShare);
        cumulativePlayerIndexSum[segment] =
            cumulativePlayerIndexSum[segment].sub(playerIndexSum);
        for (uint256 i = 0; i < rewardTokens.length; i++) {
            rewardTokenAmounts[i] = rewardTokenAmounts[i].sub(playerReward[i]);
        }
        totalIncentiveAmount = totalIncentiveAmount.sub(playerIncentive);
    }
}
```

However, since there can be precision errors when calculating each player's share percentage, updating these parameters benefits the players that haven't withdrawn their rewards yet.
These errors are being accumulated for the next withdrawal until the last one.
Hence, this constitutes a great incentive for the players to withdraw as last as possible to get the most possible rewards.

Here is an example which demonstrates the issue:

**Example #1 (multiplier = 100)**

| Withdraw No. | Player Index | % | Expected Amount | Received Amount | Diff | Total Amount |
|---|---|---|---|---|---|---|
| 1 | 100 | 3 | 1500 | | | 50000 |
| | 500 | 19 | 9500 | 9500 | 0 | |
| | 2000 | 76 | 38000 | | | |
| | Left to Contract | 2 | 1000 | | | |

| | | | | | | |
|---|---|---|---|---|---|---|
| | Cumulative Index | | 2600 | | | |
| | | | | | | |
| 2 | 100 | 4 | 1500 | 1620 | 120 | 40500 |
| | 2000 | 95 | 38000 | | | |
| | Cumulative Index | | 2100 | | | |
| | | | | | | |
| 3 | 2000 | 100 | 38000 | 38880 | 880 | 38880 |
| | Cumulative Index | | 2000 | | | |

We highly recommend using a higher precision when calculating the players share percentage to avoid this issue.

```
// Dedaub: Use higher precision for calculating the player's percentage
playerSharePercentage =
   (playerIndexSum.mul(100)).div(cumulativePlayerIndexSum[segment]);
```

Here is the above example using a higher multiplier for keeping precision to 4 decimal points instead of 2:

| Example #2 (multiplier = 10000) | | | | | | |
|---|---|---|---|---|---|---|
| Withdraw No. | Player Index | % | Expected Amount | Received Amount | Amount Diff | Total Amount |
| 1 | 100 | 384 | 1920 | | | 50000 |
| | 500 | 1923 | 9615 | 9615 | 0 | |
| | 2000 | 7692 | 38460 | | | |

| | | | | | | |
|---|---|---|---|---|---|---|
| | Left to Contract | 1 | 5 | | | |
| | Cumulative Index | | 2600 | | | |
| | | | | | | |
| 2 | 100 | 476 | 1920 | 1922 | 2 | 40385 |
| | 2000 | 9523 | 38460 | | | |
| | Cumulative Index | | 2100 | | | |
| | | | | | | |
| 3 | 2000 | 10000 | 38460 | 38463 | 3 | 38463 |
| | Cumulative Index | | 2000 | | | |

## MEDIUM SEVERITY:

[No medium severity issues]

## LOW SEVERITY:

| ID | Description | STATUS |
|---|---|---|
| L1 | Missing reentrancy guards | **RESOLVED** |

Although most Pool functions have reentrancy guards, withdraw/earlyWithdraw do not. Maybe the reason is that they are protected by the `player.withdrawn` flag. However, this flag only prevents reentering in the same function, but not cross-reentrancy between different functions.

We did not find a concrete way to exploit the lack of guard, however reentrancy attacks are hard to spot, and it is likely that a vulnerability could be introduced in future versions of the code. Hence, we recommend protecting all core methods with a guard.

## CENTRALIZATION ISSUES:

It is often desirable for DeFi protocols to assume no trust in a central authority, including the protocol's owner. Even if the owner is reputable, users are more likely to engage with a protocol that guarantees no catastrophic failure even in the case the owner gets hacked/compromised. We list issues of this kind below. (These issues should be considered in the context of usage/deployment, as they are not uncommon. Several high-profile, high-value protocols have significant centralization threats.)

| ID | Description | STATUS |
|---|---|---|
| N1 | incentiveToken can be used to block withdrawals | **RESOLVED** |

The protocol is designed to guarantee that funds can be withdrawn even if the owner gets compromised (for instance, Pool::withdraw is not pausable). However, a compromised owner could still block withdrawals by employing a malicious incentiveToken, which can be set while the game is active (if not configured during the initialization).

To achieve this, `incentiveToken::balanceOf` could be simply set to revert, either always or under specific conditions (specific users, specific time periods, etc). This would cause `Pool::withdraw` to fail, since it calls `incentiveToken::balanceOf`.

To prevent this issue, `Pool::withdraw` could be modified to tolerate any errors of `incentiveToken`.

| N2 | `emergencyWithdraw` can be used to allow admin get all `totalIncentiveAmount` tokens when no real emergency exist | **RESOLVED** |
|---|---|---|

In emergency scenarios the protocol is designed to give all incentive tokens to the owner.

```
function adminFeeWithdraw(uint256 _minAmount) external virtual onlyOwner
 whenGameIsCompleted {
    if (winnerCount == 0) {
        if (totalIncentiveAmount != 0) {
            bool success = IERC20(incentiveToken).transfer(owner(),
                                                           totalIncentiveAmount);
            if (!success) {
                revert TOKEN_TRANSFER_FAILURE();
            }
        }
    }
}
```

This happens because the `winnerCount` variable updates only when players deposit on the last segment of the game. If the game has to urgently end, then various players can be winners but the variable doesn't get updated then.

```
function _transferInboundTokenToContract(
        uint256 _minAmount,
        uint256 _depositAmount,
        uint256 _netDepositAmount
    ) internal virtual {
    if (currentSegment == depositCount.sub(1)) {
        // array indexes start from 0
        winnerCount = winnerCount.add(1);
        players[msg.sender].isWinner = true;
    }
}
```

Since giving these tokens to the owner can be desirable functionality, for a compromised owner this can be a great incentive to enable `emergencyWithdraw` before the last segment when no real emergency exists to get the whole amount of the incentive tokens.

To prevent this issue, `Pool:enableEmergencyWithdraw` could be modified to calculate the winners and update the `winnerCount` variable, but it should be considered that this will have other repercussions, such as not allowing a benevolent owner to get all incentive tokens when emergency flag is enabled, if this is the desirable functionality for emergency scenarios.

## OTHER/ ADVISORY ISSUES:

This section details issues that are not thought to directly affect the functionality of the project, but we recommend considering them.

| ID | Description | STATUS |
|---|---|---|
| A1 | Pool::getCurrentSegment has convoluted logic | **RESOLVED** |

The code in Pool::getCurrentSegment can be streamlined for better understandability. (There is a division that is guaranteed to return only 0 or 1, a complex expression that can be replaced by just depositCount, cases that can be handled slightly differently.)

Current code:

```
function getCurrentSegment() public view whenGameIsInitialized returns (uint64) {
  uint256 currentSegment;
  // logic for getting the current segment while the game is on waiting round
  if (
       waitingRoundSegmentStart <= block.timestamp &&
       block.timestamp <=(waitingRoundSegmentStart.add(waitingRoundSegmentLength))
  ) {
       uint256 waitingRoundSegment = block.timestamp.sub(waitingRoundSegmentStart)
```

```
                                          .div(waitingRoundSegmentLength);
      currentSegment = depositCount.add(waitingRoundSegment);
  } else if (block.timestamp >
            (waitingRoundSegmentStart.add(waitingRoundSegmentLength))) {
      // logic for getting the current segment after the game completes
      // (waiting round is over)
      currentSegment =
         waitingRoundSegmentStart.sub(firstSegmentStart).div(segmentLength) +
         block.timestamp.sub(
              (waitingRoundSegmentStart.add(waitingRoundSegmentLength))
         ).div(segmentLength) + 1;
  } else {
      // logic for getting the current segment during segments that allows
      // depositing (before waiting round)
      currentSegment =
         block.timestamp.sub(firstSegmentStart).div(segmentLength);
  }
  return uint64(currentSegment);
}
```

We propose (but the developers should verify) that this is equivalent to:

```
function getCurrentSegment() public view whenGameIsInitialized returns (uint64) {
  uint256 currentSegment;
  uint256 endOfWaitingRound =
                      waitingRoundSegmentStart.add(waitingRoundSegmentLength);
  if (waitingRoundSegmentStart <= block.timestamp &&
       block.timestamp < endOfWaitingRound )
  ) {
      currentSegment = depositCount;
  } else if (block.timestamp >= endOfWaitingRound) {
     // logic for getting the current segment after the game completes
     // (waiting round is over)
     currentSegment = depositCount + 1 +
              block.timestamp.sub(endOfWaitingRound).div(segmentLength);
  } else {
      // logic for getting the current segment during segments that allows
      // depositing (before waiting round)
     currentSegment = block.timestamp.sub(firstSegmentStart).div(segmentLength);
```

```
  }
  return uint64(currentSegment);
}
```

| A2 | Initialization needs to be complex | DISMISSED |
|---|---|---|

It is not entirely clear how the contracts will get initialized so that the right ownership structure is put in place. It seems that this can only happen via `transferOwnership` of the standard Ownable functionality.

Specifically, the strategies have some `onlyOwner` functions (e.g., `earlyWithdraw`) whose clear intent is that they only be callable by the Pool contract. E.g., in NoExternalStrategy:

```
function earlyWithdraw(
       address _inboundCurrency,
       uint256 _amount,
       uint256 _minAmount
) external override onlyOwner {
       _transferInboundTokenToPool(_inboundCurrency, _amount);
}
```

The last function call transfers tokens to the `msg.sender`, which is the contract's owner. But the name of the function clearly states that the tokens go to the pool. It is not clear how the pool has become the owner, since it's not done at construction and the pool has no way to construct the strategy.

| A3 | Logic that limits transfer amounts can be factored out | RESOLVED |
|---|---|---|

This code fragment occurs nearly identically in several places in the code (of Pool):

```
if (isTransactionalToken) {
       if (payout > address(this).balance) {
              payout = address(this).balance;
       }
       (bool success, ) = msg.sender.call{ value: payout }("");
       if (!success) {
              revert TRANSACTIONAL_TOKEN_TRANSFER_FAILURE();
```

```
        }
} else {
        if (payout > IERC20(inboundToken).balanceOf(address(this))) {
                payout = IERC20(inboundToken).balanceOf(address(this));
        }
        bool success = IERC20(inboundToken).transfer(msg.sender, payout);
        if (!success) {
                revert TOKEN_TRANSFER_FAILURE();
        }
}
```

Either the entire fragment or each branch individually can be factored out and reused.

| A4 | Constructor check may be too loose | RESOLVED |
|---|---|---|

The check for compatibility of underlying assets in the Pool constructor may be too loose.

```
if (_underlyingAsset != address(0) && _underlyingAsset != _inboundCurrency &&
    !_isTransactionalToken) {
      revert INVALID_INBOUND_TOKEN();
}
```

The check will succeed if `_underlyingAsset` (i.e., the strategy's asset) is `address(0)` even if `_inboundCurrency` and `_isTransactionalToken` disagree.

| A5 | Opportunities for gas savings | RESOLVED |
|---|---|---|

There are points in the code where gas savings can be realized, mostly by caching the results of storage loads. Examples include:

Pool::_transferInboundTokenToContract

```
for (uint256 i = 0; i <= players[msg.sender].mostRecentSegmentPaid; i++) {
  // Dedaub: guaranteed to be currentSegment?
```

Pool::adminFeeWithdraw

```
if (address(rewardTokens[i]) != address(0)) {
   if (adminFeeAmount[i + 1] != 0) {
```

```
        bool success = rewardTokens[i].transfer(owner(), adminFeeAmount[i + 1]);
      // Dedaub: could cache adminFeeAmount[i+1], also rewardTokens[i]
```

| A6 | Hard-coded use of WMatic | RESOLVED |
|---|---|---|

Reward token is assumed to be WMatic (or at least have the same interface) in some parts of AaveStrategy. Is this reasonable?

Also, hard-coded constants ("magic constants") in the code should be best avoided for maintainability, at least by giving the constant a high-level name.

```
function invest(address _inboundCurrency, uint256 _minAmount) external payable
override onlyOwner {
 if (_inboundCurrency == address(0) || _inboundCurrency == address(rewardToken)) {
     if (_inboundCurrency == address(rewardToken)) {  // Dedaub: the only case?
        // unwraps WMATIC back into MATIC
        WMatic(address(rewardToken)).withdraw(IERC20(_inboundCurrency).
                                              balanceOf(address(this)));
     }
     // Deposits MATIC into the pool
     wethGateway.depositETH{ value: address(this).balance }
      (address(lendingPool), address(this), 155); // Dedaub: Magic constant
 } else { …
```

| A7 | Native token receive functions are too liberal | RESOLVED |
|---|---|---|

It is not clear why the `receive` functions are not more closely guarded. For instance, for strategies, what is the purpose of the `receive` function? (The strategy can receive native tokens via payable functions, but receiving them silently only complicates the accounting obligations.)

```
receive() external payable {}
```

Similarly, for the Pool contract, the only caller of the `receive` function in the course of the game should be the strategy. The `receive` function can be protected to be only

callable by the strategy, again, to remove the need for thinking about unpredictable balance changes in the protocol.

```
receive() external payable {
       if (!isTransactionalToken) {
              revert INVALID_TRANSACTIONAL_TOKEN_AMOUNT();
       }
}
```

| A8 | Compiler bugs | **INFO** |
|---|---|---|

The code is compiled with Solidity 0.8.7 or higher. For deployment, we recommend no floating pragmas, i.e., a specific version, so as to be confident about the baseline guarantees offered by the compiler. Version 0.8.7, in particular, has some known bugs, which we do not believe to affect the correctness of the contracts.

## DISCLAIMER

The audited contracts have been analyzed using automated techniques and extensive human inspection in accordance with state-of-the-art practices as of the date of this report. The audit makes no statements or warranties on the security of the code. On its own, it cannot be considered a sufficient assessment of the correctness of the contract. While we have conducted an analysis to the best of our ability, it is our recommendation for high-value contracts to commission several independent audits, as well as a public bug bounty program.

## ABOUT DEDAUB

Dedaub offers technology and auditing services for smart contract security. The founders, Neville Grech and Yannis Smaragdakis, are top researchers in program analysis. Dedaub's smart contract technology is demonstrated in the contract-library.com service, which decompiles and performs security analyses on the full Ethereum blockchain.