Dedaub

Security Technology for Smart Contracts

# GoodGhosting

## Smart Contract Security Assessment

Date: July 06, 2021

## Abstract

Dedaub was commissioned to perform a security audit of the GoodGhosting contracts (specifically: MerkleDistributor.sol, GoodGhosting.sol, GoodGhostingPolygon.sol, GoodGhostingPolygonWhitelisted.sol), at commit hash `691ae203636d0f3ca4419fa81de26549bd754f24`. The audit is explicitly about the code. Two auditors worked over this codebase for 3 days and consulted other engineers on-demand for specialized expertise.

## Setting and Caveats

The audited code base is of small size, at around 700LoC (excluding test and interface code). The audit focused on security, establishing the overall security model and its robustness, and also crypto-economic issues. Functional correctness (e.g., that the calculations are correct) was a secondary priority. 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.

## Architecture and Recommendations

The contracts implement an investment game, incentivizing players to maintain an investment schedule. The game is rather “friendly”, always allowing the withdrawal of principal at the end of the game, even for non-winning players who stopped maintaining their investment schedule. Withdrawal before the end of the game incurs a penalty, which is distributed to the winning players.

The code is mature and well-tested.

The design is largely decentralized, although there are some centralization concerns that players should be aware of, if they do not trust the deployer/owner of the game. The greatest one is that the owner collects all earnings if there are no winners. Therefore, the owner can, for instance, pause the game right before the winning (penultimate) segment, so that no winners can be declared, and claim all earnings in as little as two segments later. The players still have the ability to withdraw their principal. In a real-world setting, it seems likely that this issue will not be a concern for players: if there are several games and very few, known, owners, the loss of reputation for an owner will be much more important than the earnings of a single game.

## Vulnerabilities and 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>1) User or system funds can be lost when third party systems misbehave.<br>2) DoS, under specific conditions.<br>3) Part of the functionality becomes unusable due to programming error. |
| **Low** | Examples:<br>1) Breaking important system invariants, but without apparent consequences.<br>2) Buggy functionality for trusted users where a workaround exists.<br>3) Security issues which may manifest when the system evolves. |

Issue resolution includes "dismissed", by the client, or "resolved", per the auditors. Resolved items are per commit hash 68764656a8e08c2be8fbe1e816307d4a0ac79b56.

## Critical Severity

*[No critical severity issues]*

## High Severity

| Id | Description | Status |
|---|---|---|
| **H1** | **Winning player may receive no bonus** | **Resolved** |

In `GoodGhosting::withdraw` and `GoodGhostingPolygon::withdraw` it is possible that the bonus of a winning player is calculated before `redeemFromExternalPool()` is ever called for this game.

```
    uint256 payout = player.amountPaid;
    if (player.mostRecentSegmentPaid == lastSegment.sub(1)) {
        // Player is a winner and gets a bonus!
        payout = payout.add(totalGameInterest.div(winners.length));
    }

    // First player to withdraw redeems everyone's funds
    if (!redeemed) {
        redeemFromExternalPool();
    }
```

But `redeemFromExternalPool()` redeems the accumulated funds from the external pool along with the accrued interest which is shared as bonus to the winners. Consequently, in case that the first player to call withdraw is a winning one, then this player will receive no bonus.

We suggest that `redeemFromExternalPool()` be called before a player's bonus calculation:

```
    // First player to withdraw redeems everyone's funds
    if (!redeemed) {
        redeemFromExternalPool();
    }

    uint256 payout = player.amountPaid;
```

```
    if (player.mostRecentSegmentPaid == lastSegment.sub(1)) {
        // Player is a winner and gets a bonus!
        payout = payout.add(totalGameInterest.div(winners.length));
    }
```

## Medium Severity

*[No medium severity issues]*

## Low Severity

| Id | Description | Status |
|---|---|---|
| **L1** | **No tolerance to Aave losses** | **Resolved (largely)** |

Although Aave aTokens should always be redeemable for (at least an equal amount of) their underlying tokens, the developers could consider refactoring in order to have some tolerance to Aave losses (e.g., due to hacks). Specifically, the computation (in `GoodGhosting::redeemFromExternalPool` and `GoodGhostingPolygon::redeemFromExternalPool`)

```
uint256 grossInterest = totalBalance.sub(totalGamePrincipal);
```

contains an implicit `require` in the `sub`. This means that the transaction will revert if the balance after an Aave withdrawal is lower than the capital invested. It is not clear that this is a condition that should cause a revert. There may be funds that can be withdrawn, although lower than those invested originally.

| Id | Description | Status |
|---|---|---|
| **L2** | **Matic rewards also not claimable if there are no admin fees** | **Resolved** |

In `GoodGhostingPolygon::adminFeeWithdraw` the following require seems too strict:

```
require(adminFeeAmount > 0, "No Fees Earned");
```

Although in regular GoodGhosting there is no meaning in calling this function in case of no fees collected (mainly in case of a fee-free game setup), in the GoodGhosting Polygon version this check also prevents the execution of the second part of the function, which is claiming Matic rewards. We recommend turning the check into an `if`.

## Other/Advisory Issues

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

| Id | Description | Status |
|---|---|---|
| **A1** | **Function ordering inside contracts** | **Resolved** |
| Consider adopting the official style guide for function ordering within a contract.<br>In order of priority:<br>external > public > internal > private and view > pure within the same visibility group.<br>https://docs.soliditylang.org/en/v0.4.24/style-guide.html#order-of-functions | | |
| **A2** | **Non-intuitive variable name** | **Resolved** |
| In `GoodGhostingPolygon::redeemFromExternalPool` variable `amount` holds the accrued rewards of the game. We suggest this variable be renamed to `rewardsAmount` for clarity. | | |
| **A3** | **Redundant check** | **Dismissed [Clarity of messages more important]** |

In `GoodGhosting::makeDeposit` the first of the following requires

```
//check if current segment is currently unpaid
require(
    players[msg.sender].mostRecentSegmentPaid != currentSegment,
    "Player already paid current segment"
);

// check if player has made payments up to the previous segment
require(
    players[msg.sender].mostRecentSegmentPaid == currentSegment.sub(1),
        "Player didn't pay the previous segment - game over!"
);
```

is redundant, as it is covered by the latter one. We suggest that it be removed in order to save gas. If this is done, the error message should also change for clarity, so as to reflect any possible error case.

| A4 | Inconsistent reference to owner | Resolved |
|---|---|---|

In owner-only function `GoodGhostingPolygon::adminFeeWithdraw` the owner of the contract is written both as `owner()` and `msg.sender`.

```
emit AdminWithdrawal(owner(), totalGameInterest, adminFeeAmount);

require(
    // Dedaub: owner as owner()
    IERC20(daiToken).transfer(owner(), adminFeeAmount),
    "Fail to transfer ER20 tokens to admin"
);
if (rewardsPerPlayer == 0) {
    uint256 balance = IERC20(matic).balanceOf(address(this));
    require(
        // Dedaub: owner as msg.sender
        IERC20(matic).transfer(msg.sender, balance),
        "Fail to transfer ERC20 tokens on withdraw"
    );
}
```

We suggest that `msg.sender` be altered to `owner()` for consistency and clarity (or the converse, in case the gas savings are considered to be worth it).

| A5 | Code simplification/gas savings opportunity | Resolved |
|---|---|---|

In `GoodGhosting::_joinGame` the code below

```
if (!canRejoin) {
   iterablePlayers.push(msg.sender);
}
require(iterablePlayers.length <= maxPlayersCount, "Reached max quantity
of players allowed");
```

can change to:

```
if (!canRejoin) {
   iterablePlayers.push(msg.sender);
   require(iterablePlayers.length <= maxPlayersCount, "Reached max
quantity of players allowed");
```

```
}
```

The condition in the `require` clause can only be violated if the `if` statement executes. The change will result in slightly simpler and more gas-efficient code.

| A6 | Small remainder amounts can stay in the contract | Dismissed [already mitigated as much as possible] |
|---|---|---|

Due to the integer division in the two `withdraw` functions (GoodGhosting and GoodGhostingPolygon), small amounts can remain in the contract:

```
payout = payout.add(totalGameInterest.div(winners.length));
```

| A7 | Are Aave referral codes still active? | Dismissed [kept even if currently unnecessary] |
|---|---|---|

GoodGhosting::_transferDaiToContract contains an Aave deposit with a referral code (155). It is unclear whether this is still useful/used:

```
lendingPool.deposit(address(daiToken), segmentPayment, address(this), 155);
```

| A8 | Compiler known issues | Info |
|---|---|---|

The contracts were compiled with the Solidity compiler `v0.6.11` which, at the time of writing, has some known bugs (ABIDecodeTwoDimensionalArrayMemory, keccakCaching, EmptyByteArrayCopy, DynamicArrayCleanup). We believe the contract to be unaffected: there is no usage of `abi.decode`, no keccak hashes of statically known length, no copies of empty byte memory/calldata arrays to storage and the base types of the dynamic arrays are > 16 bytes.

## 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 status 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.