CertiK Audit Report for IEarn Finance

CertiK Eng Team
Feb 19, 2020
Version 1.1.0

Executive Summary

We found a few minor vulnerabilities in the contract, as well as some possible improvements from gas & control-flow perspectives. Most importantly, we believe to have identified a major vulnerability, which under quite common situations could temporarily block users from withdrawing all of their funds. Additionally, while not mentioned in this report, we'd like to bring the client's attention to the recent strange behavior of the bZx contracts. Being beyond the scope of the current audit, the client has to make a call whether they want to include that provider in their system. Most of our analysis is directed to the functioning of the subject contract, given the safety of the provider systems.

Review Notes

Items are labeled `CRITICAL`, `MAJOR`, `MINOR`, `INFO`, `DISCUSSION` (in decreasing significance).

Scope of work

The scope of the work was the contract yDAI with its full Solidity inheritance chain and all libraries. The file those contracts can be found in, and the file that all line references refer to, is:

- https://github.com/iearn-finance/itoken/blob/36360b8d5ad43c1ca0685b06c4f6e9b6c0d58cb5/contracts/YDAIv2.sol

Orthogonal contracts and systems are not part of the scope, namely:

- providers
  - Compound
  - Fulcrum
  - dYdX
  - Aave
- DAI system

Assumptions

Our audit was predicated upon the following assumptions.

Providers

The following is pseudocode. We use `T` for the native provider token.

- Compound
  - balance (DAI)
    - `c.balanceOf (this) * c.exchangeRateStored() / 1e18`
  - supply (DAI)
    - `a -> c.mint (a)`
    - returns `uint (0)` on success
  - withdraw (T)
    - `a -> c.redeem (a)`
    - returns `uint (0)` on success
- Fulcrum
  - balance (DAI)
    - `f.assetBalanceOf (this)`
    - MINOR it should be noted that this function does not currently exist in the abi of the contract.
    - an implicit invariant is assumed: if `f.balanceOf (this)` is 0, then the above is also 0
  - supply (DAI)
    - `a -> f.mint (this, a)`
    - should return > 0 on success
  - withdraw (T)
    - `a -> f.burn (this, a)`
    - should return > 0 on success
- dYdX
  - balance (DAI)

        - `d.getAccountWei (Info (this, 0), dToken)`
        - returns a struct `s` with `sign` and `value`
        - `sign` is assumed to always be `true`
    - supply (DAI)

```
a -> d.operate ([Info (this, 0)], [ActionArgs (
  Deposit,
  0,
  AssetAmount (true, Wei, Delta, a),
  dToken,
  0,
  this,
  0,
  []
)])
```

- dYdX
    - withdraw (DAI)

```
a -> d.operate ([Info (this, 0)], [ActionArgs (
  Withdraw,
  0,
  AssetAmount (false, Wei, Delta, a),
  dToken,
  0,
  this,
  0,
  []
)])
```

- Aave
    - balance (DAI)
        - `aaveToken.balanceOf (this)`
    - supply (DAI)
        - `a -> aave.getLendingPool().deposit (token, a, 0)`
    - withdraw (DAI)
        - `a -> aaveToken.redeem (a)`

INFO It is assumed none of these contracts are malicious or exploited. Since the constructor function sets an infinite approval (w.r.t. integer bounds) then the contract could steal all DAI held by yDAI. It should be noted that the attack vector here is the same as for all other participants in these systems.

We have confirmed that both the public functions and the internal ones (prefixed by `_`) meet these assumptions.

Analysis

Functions

- fallback
  - INFO Consider removing this. Any ether sent to this address will be forever stuck.
- withdraw & redeem
  - allows to withdraw DAI. Updates pool.
  - If contract doesn't have enough DAI, withdraws the remainder from the current provider. This can be divided into two cases:
    - aave and dydx allow to withdraw DAI directly. For the call to succeed, a check is made about balances (L620, L624).
      - MAJOR This check may not always succeed, leading to balances not be withdrawable. Let's rewrite the check:
        - $bD \geq a$
        - $bD \geq r - b$
        - $bD \geq \frac{sh}{tS}(pool) - b$
        - $bD \geq \frac{sh}{tS}(b + bD + bA + bF + bC) - b$
        - $bD + b \geq \frac{sh}{tS}(b + bD + bA + bF + bC)$
        - Now, since `supplyXXX` functions are public, it can happen that balances in the other tokens make the RHS larger than LHS. In particular, here are a few situations when that can occur:
          - user deposits & supplies a different provider than the current provider.

            - the balances in the current provider lead to a negative return (the assumption that the providers provide a non-decreasing return is not made, so that is this case).
            - another user supplies a different provider and the return there is greater
            - a malicious user supplies a different provider and transfer its tokens to this contract to simulate it having a higher return.
        - This situation will last until a new provider is set.
        - This bug is all the more severe that it can't be resolved by simply withdrawing small amounts one by one. Partial withdrawals will work, but there will always be the leftover that won't work.
        - Recommendation: make `supplyXXX` functions `internal`.
    - we don't assume fulcrum and compound allow to withdraw DAI directly, so instead we have `withdrawSomeXXX`
        - MAJOR these functions also make a check about the balance, so they are vulnerable in the same vector as described above
        - MINOR These functions add 1 to the amount of tokens being withdrawn (L599, L611). We understand the motivation for this - otherwise an insufficient amount of DAI could be withdrawn and the yDAI withdraw would fail. However, this itself can also lead to a withdrawal blocks. If there is only one balance holder, then `_shares = _totalSupply`, hence `r = bT` and we will be attempting to withdraw $b + 1$ tokens. However, any `_amount` smaller will still be withdrawable.
    - INFO Consider calling `_burn`.
- supplyXXX
    - INFO Consider making these internal. This will resolve the Major above, and also lead to a nice invariant: the contract will ever hold balances (up to truncated division) in just one provider.

Overflows

We have not found any situations where overflow would occur. That is because of the use of `safeMath`. We also haven't found situations where overflow would occur inside the `safeMath` library, leading to a revert and a vulnerability. This is because all math (a total of 21 times) is performed on one of these:

- yDAI balances
- numbers from providers
  - native provider token balances
  - provider DAI balances
  - exchange rates
- direct DAI balances

Truncated division

The EVM performs floor division on integers. This means the result may differ from the mathematical one by $0 \geq D \geq \frac{den-1}{den}$ for a denominator den. Division occurs nine times, but we didn't find events where this leads to unintended behavior.

Game-theoretic security

We haven't found any vulnerabilities related to the ordering & inputs of message calls. This is mainly because all contracts called (DAI and providers) are assumed to not be malicious (e.g. not calling back into this one). We haven't found instances of front-running or other game-theoretic vulnerabilities.

Control-flow & Gas savings

INFO We think there are a lot of opportunities for gas savings in the system. We view the following as passing the trade-off between gas and security:

- rebalance & _rebalance

    - a call to `balance()` is made 5 times. Consider making it once and storing it as a local variable.
- deposit & invest
    - consider removing L387 & L704

Appendix: Overview Graph