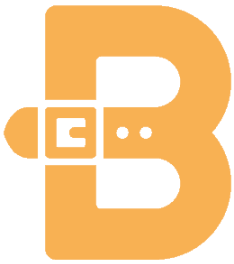
Belt Finance Audit

Smart Contract Security Assessment

Nov 1, 2021



ABSTRACT

Dedaub was commissioned to perform a security audit on the Belt Finance yield farming contracts. We received the code base in the form of an archive file on Oct. 8, 2021. The contracts in scope comprise strategies, investment tokens, and investment entry points. The primary contracts (non-test, non-interface, non-mock) are listed below:

```
BeltToken.sol
earn/MasterBelt.sol
earnV2/Assistant.sol
earnV2/MasterOrbit.sol
utils/BeltProxy.sol
utils/Timelock.sol
utils/UnwrapperHT.sol
utils/Unwrapper.sol
earn/strategies/VaultBPool.sol
earn/strategies/VaultCakePool.sol
earnV2/strategies/StrategyAlt.sol
earnV2/strategies/Strategy.sol
earnV2/strategiesV2/StrategyV2Alt.sol
earnV2/strategiesV2/StrategyV2.sol
earnV2/tokens/MultiStrategyTokenImpl.sol
earnV2/tokens/MultiStrategyToken.sol
earnV2/tokens/MultiStrategyTokenStorageBSC.sol
earnV2/tokens/MultiStrategyTokenStorageHECO.sol
earnV2/tokens/MultiStrategyTokenStorage.sol
earnV2/tokens/SingleStrategyToken2.sol
earnV2/tokens/SingleStrategyTokenImpl2.sol
earnV2/tokens/SingleStrategyTokenImpl.sol
earnV2/tokens/SingleStrategyToken.sol
earnV2/tokens/SingleStrategyTokenStorageBSC.sol
earnV2/tokens/SingleStrategyTokenStorageHECO.sol
earnV2/tokens/SingleStrategyTokenStorage.sol
earnV2/tokens/StrategyToken.sol
earnV2/tokensV2/BeltStrategyTokenV2.sol
earnV2/tokensV2/BeltStrategyTokenV2Storage.sol
earnV2/tokensV2/StrategyTokenV2.sol
earnV2/tokensV2/VoidStrategyTokenStorageV2.sol
earnV2/tokensV2/VoidStrategyTokenV2.sol
earnV2/strategies/alpaca/StrategyAlpacaImpl.sol
earnV2/strategies/alpaca/StrategyAlpaca.sol
earnV2/strategies/alpaca/StrategyAlpacaStorageBSC.sol
earnV2/strategies/alpaca/StrategyAlpacaStorage.sol
earnV2/strategies/alpha/StrategyAlphaImpl.sol
earnV2/strategies/alpha/StrategyAlpha.sol
earnV2/strategies/alpha/StrategyAlphaStorageBSC.sol
```

```
earnV2/strategies/alpha/StrategyAlphaStorage.sol
earnV2/strategies/channels/StrategyChannelsImpl.sol
earnV2/strategies/channels/StrategyChannels.sol
earnV2/strategies/channels/StrategyChannelsStorageHECO.sol
earnV2/strategies/channels/StrategyChannelsStorage.sol
earnV2/strategies/ellipsis/StrategyEllipsisImpl.sol
earnV2/strategies/ellipsis/StrategyEllipsis.sol
earnV2/strategies/ellipsis/StrategyEllipsisStorageBSC.sol
earnV2/strategies/ellipsis/StrategyEllipsisStorage.sol
earnV2/strategies/filda/StrategyFildaImpl.sol
earnV2/strategies/filda/StrategyFilda.sol
earnV2/strategies/filda/StrategyFildaStorageHECO.sol
earnV2/strategies/filda/StrategyFildaStorage.sol
earnV2/strategies/lendhub/StrategyLendHubImpl.sol
earnV2/strategies/lendhub/StrategyLendHub.sol
earnV2/strategies/lendhub/StrategyLendHubStorageHECO.sol
earnV2/strategies/lendhub/StrategyLendHubStorage.sol
earnV2/strategiesV2/belt/StrategyBeltV2.sol
earnV2/strategiesV2/belt/StrategyBeltV2Storage.sol
earnV2/strategiesV2/fortube/StrategyFortubeV2.sol
earnV2/strategiesV2/fortube/StrategyFortubeV2StorageBSC.sol
earnV2/strategiesV2/fortube/StrategyFortubeV2Storage.sol
earnV2/strategiesV2/venus/StrategyVenusV3.sol
earnV2/strategiesV2/venus/StrategyVenusV3StorageBSC.sol
earnV2/strategiesV2/venus/StrategyVenusV3Storage.sol
earnV2/strategiesV2/void/StrategyVoidV2.sol
earnV2/strategiesV2/void/StrategyVoidV2Storage.sol
```

The audit also examined any other functionality highly related to the included contracts.

SETTING & CAVEATS

The audited code base is of significant apparent size, at nearly 14KLoC. However, large amounts of code are inlined verbatim. Once verbatim copies of entire contracts are eliminated, the size of the code base drops to around 9KLoC. There still remains some duplication of code and structure among strategies, but this is at the level expected from a multi-investment yield farming protocol.

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.

ARCHITECTURE & HIGH-LEVEL RECOMMENDATIONS

The project's architecture minimizes the attack surface, with few untrusted entry points, mainly for external investors. All "internal" levels of functionality (e.g., strategies) are implemented with uniform patterns (e.g., proxy+ impl+storage) and uniformly tokenized. This makes for a clean and modular architecture.

There are **two major elements** in our high-level recommendations. These are listed separately from the subsequent "vulnerabilities and functional issues" since they do not concern directly the smart contract code but its setup and design.

- We have received no testing code and the repository does not seem to have an actively maintained test suite. This should be remedied. Even though the contracts are already deployed, thorough testing is invaluable and regressions are easy to introduce. Furthermore, deployment in chains with less tooling support (e.g., HECO) puts an even heavier burden on local testing.
- The functionality has serious centralization threats: the governance/owner of the protocol (i.e., the owner account of MasterBelt) can cause significant damage or steal funds. We did not consider "attack-by-owner" scenarios in detail, since the assumption for a prominent protocol is that the owner's interests align with the well-being of the protocol. But it is clear that such attacks exist. For example, two attacks-by-owner are:

    a) The owner can add a new pool but reuse a past strategy, yet specify a different "want" token. This can create users with an inconsistent number of shares, permitting the draining of other users' funds.

    b) The owner can add a new pool with BELT as the "want" token, allowing users to withdraw all of the BELT holdings of MasterBelt.

  This problem is exacerbated by having the owner of the current MasterBelt (BSC:0xD4BbC80b9B102b77B21A06cb77E954049605E6c1) be an EOA (BSC:0x7111D0F651A331BC2b9eeFCFE56D8A03F92601a1). It is highly recommended that ownership be transferred to a multi-sig wallet setup (e.g., a 3-of-5 with independent holders).

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: 01) User or system funds can be lost when third party systems misbehave. 02) DoS, under specific conditions. 03) Part of the functionality becomes unusable due to programming error.
LOW	Examples: 01) Breaking important system invariants, but without apparent consequences. 02) Buggy functionality for trusted users where a workaround exists. 03) Security issues which may manifest when the system evolves.

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

CRITICAL SEVERITY

[No critical severity issues]

HIGH SEVERITY:

ID	Description	STATUS
H1	Untrusted callers can force swaps on AMMs	**RESOLVED** (Mitigated by requiring minimum amount and trusted caller. Responsibility moved to caller.)

The `earn()` method of nearly all strategies allows calling by an untrusted external account. However, `earn()` swaps the strategy's earnings using an on-chain AMM (PancakeSwap currently, in the future possibly also other Uniswap-based ones). This permits a sandwich attack, where an attacker can distort the price of the earning token to make it appear worthless, and then restore the price, receiving most of the swapped funds as profit.

For instance, the code in the Alpaca strategy is:

```
function earn() external whenNotPaused onlyEOA {
       FairLaunch(fairLaunchAddress).harvest(poolId);

       uint256 earnedAmt = AlpacaToken(alpacaAddress).balanceOf(address(this));
       earnedAmt = buyBack(earnedAmt);

       if (alpacaAddress != wantAddress) {
              IPancakeRouter02(uniRouterAddress).swapExactTokensForTokens(
              earnedAmt,
              0,
              alpacaToWantPath,
              address(this),
              now.add(600)
              );
       }
  ...
}

function buyBack(uint256 _earnedAmt) internal returns (uint256) {
       if (buyBackRate == 0 && buyBackPoolRate == 0) {
              return _earnedAmt;
```

```
        }

        uint256 buyBackAmt =
          _earnedAmt.mul(buyBackRate.add(buyBackPoolRate)).div(buyBackRateMax);

        IPancakeRouter02(uniRouterAddress).swapExactTokensForTokens(
                buyBackAmt,
                0,
                alpacaToBELTPath,
                address(this),
                now + 600
        );
    ...
}
```

That is, the earnings (in the Alpaca token) are swapped both to BELT and to the "want" token. Such swaps can be sandwiched by an attacker. For a sandwich attack to be profitable, the swapped amount should be more than the 0.2% of the pool liquidity, where 0.2% is the current fee for PancakeSwap.

Although BSC is considered less susceptible to sandwich attacks, they are possible, via collaboration with validators. Current explorers show several sandwiching transactions. The only protection in the Belt code is the `onlyEOA` modifier, which currently prevents an atomic call from simple contract code, but not an atomic attack by a validator. Therefore, for the code to actually be safe, there should be vigilance in calling `earn()` preemptively before the earnings become significant enough to attack.

In the future, there may be more threats in the above code, e.g., the `onlyEOA` modifier not being sufficient to guarantee that there is no prior atomic manipulation.

MEDIUM SEVERITY:

ID	Description	STATUS
M1	Inaccurate accounting regarding BELT rewards	**DISMISSED**

Upon depositing an amount through `MasterBelt::deposit` (or `MasterOrbit::deposit`), rewards accrued up to that point are transferred to the user (depositor), so that the upcoming reward calculation considers the user's updated balance. For the transfer, a custom safeBELTTransfer function is called which bounds the transferred amount by the BELT balance of the contract. However, even if the whole desired reward amount is not actually transferred to the user, the accounting zeroes out the pending rewards for this user.

```
if (user.shares > 0) {
    uint256 pending =
    user.shares.mul(pool.accBELTPerShare).div(1e12).sub(user.rewardDebt);
    if (pending > 0) {
        // Dedaub: this should return the amount that was actually transferred so
that the rewardDebt calculation in the end is precise.
        // Dedaub: otherwise the user receives less total rewards
        safeBELTTransfer(msg.sender, pending);
     }
}
//...
// Dedaub: updating user.rewardDebt to reflect zero pending rewards
user.rewardDebt = user.shares.mul(pool.accBELTPerShare).div(1e12);
```

Same issue exists in `MasterBelt::withdraw` (or `MasterOrbit::withdraw`).

It is not clear whether this scenario is realistic (since BELT is regularly minted according to the pools' allocation points). In any case we recommend that the code be updated regarding this issue for clarity and ease of maintenance.

M2	Withdrawal fees twice applied	**RESOLVED**

In StrategyEllipsis.sol withdrawal fees are being applied twice:

```
function withdraw(uint256 _wantAmt)
       external
       onlyOwner
       nonReentrant
       returns (uint256)
{
       _wantAmt = _wantAmt.mul(withdrawFeeDenom.sub(withdrawFeeNumer)
           ).div(withdrawFeeDenom);

       _withdraw(_wantAmt);
       // ...
```

```
}

function _withdraw(uint256 _wantAmt) internal {

        // Dedaub: fees have already been extracted in withdraw()
        _wantAmt = _wantAmt.mul(withdrawFeeDenom.sub(withdrawFeeNumer)
            ).div(withdrawFeeDenom);

        // ...
}
```

LOW SEVERITY:

ID	Description	STATUS
L1	Implicit requirement in MultiStrategyToken	**DISMISSED**

The implementation of MultiStrategyToken considers that all of the included strategies (SingleStrategyToken) support the same underlying token, which is also the underlying token of the MultiStrategyToken itself. However, there is no such check or requirement in the corresponding contracts, which makes the code less readable and clear but also potentially error prone. For example, the deployment of a MultiStrategyToken with an inappropriate strategy is possible. In such a case the protocol would normally execute transactions without emitting any errors until someone actually notices and users' funds would probably be at risk (e.g., an included strategy supports an undervalued token compared to the appropriate one). We recommend that appropriate require()ments be added in MultiStrategyToken::constructor.

OTHER/ ADVISORY ISSUES:

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

ID	Description	STATUS
A1	Dead code	**DISMISSED**

There are a number of dead code cases, such as duplicate sanity checks or unused functions.

- Redundant checks

  In `MasterBelt::updatePool`:

```
if (block.number <= pool.lastRewardBlock) {
    return;
}
// ...
uint256 multiplier = getMultiplier(pool.lastRewardBlock, block.number);
// Dedaub: redundant check, already checked above
if (multiplier <= 0) {
    return;
}
```

  Similarly in `MasterOrbit::updatePool`:

```
if (mined <= pool.lastMined) {
    return;
}
// ...
uint256 multiplier = getMultiplier(pool.lastRewardBlock, block.number);
// Dedaub: redundant check, already checked above
if (multiplier <= 0) {
    return;
}
```

  In `MasterBelt::withdraw`:

```
if (sharesRemoved > user.shares) {...}
else {
  user.shares = user.shares.sub(sharesRemoved);
  // Dedaub: no need for safe arithmetic
}
```

- Unused functions

In MultiStrategyTokenImpl.sol functions

```
function addStrategy(address strategyAddress) internal
function removeStrategy(address strategyAddress) internal
```

are internal but never called (as far as we can tell from the contracts provided).

Similarly in SingleStrategy.sol:

```
function _setNewStrategy(address newStrategyAddr) internal
```

- Unused contracts

It is not clear whether the Timelock contract is currently being used.

A2	Avoid code duplication	**DISMISSED**

Here are some easy-to-implement suggestions for decreasing code duplication within the contracts making them more readable and easier to maintain.

1. Add more internal functions.

- In MasterBelt.sol, `BELTReward` amount is calculated more than once

```
uint256 BELTReward =
multiplier.mul(BELTPerBlock).mul(pool.allocPoint).div(totalAllocPoint);
```

Consider having an internal function for this calculation.
Same holds for `pending` amount calculation:

```
uint256 pending =
user.shares.mul(pool.accBELTPerShare).div(1e12).sub(user.rewardDebt);
```

- Also for `accBeltPerShare`, which could be calculated by an internal pure function:

```
pool.accBELTPerShare =
pool.accBELTPerShare.add(BELTReward.mul(1e12).div(sharesTotal));
```

- And for a pool's `wantLockedTotal` and `sharesTotal` amount:

```
uint256 wantLockedTotal = IStrategy(poolInfo[_pid].strat).wantLockedTotal();

uint256 sharesTotal = IStrategy(poolInfo[_pid].strat).sharesTotal();
```

2. Better placement of sanity checks.

- In MultiStrategyToken.sol and BeltStrategyTokenV2.sol functions deposit() and depositBNB() check the `depositPaused` variable.

```
require(!depositPaused, "deposit paused");
```

The check could be placed once in the internal function _deposit() instead, which is called in any case.

3. Turn commonly used require()ments to modifiers.

- In SingleStrategyToken.sol checks for authorised msg.sender are made in numerous functions

```
require(msg.sender == govAddress || msg.sender == owner(), "Not authorized");
```

Consider making an onlyAuthorised modifier.

Similarly for only-government actions in numerous contracts (VaultBPool.sol, VaultCakePool.sol, StrategyAlpacaImpl.sol, StrategyAlphaImpl.sol, StrategyChannelsImpl.sol, StrategyEllipsisImpl.sol, StrategyFildaImpl.sol, StrategyLendHubImpl.sol, StrategyBeltV2.sol, StrategyFortubeV2.sol, StrategyVenusV3.sol, StrategyVoidV2.sol)

```
require(msg.sender == govAddress, "!gov");
```

- Modifiers for deposit/withdraw paused/unpaused (SingleStrategyTokenImpl, SingleStrategyTokenImpl2, MultiStrategyTokenImpl, BeltStrategyTokenV2, VoidStrategyTokenV2)

```
require(!depositPaused, "deposit paused")
require(depositPaused, "deposit not paused")
require(!withdrawPaused, "withdraw paused")
require(withdrawPaused, "withdraw not paused")
```

- Modifier for BNB token requirement

```
require(isWbnb, "not bnb");
```

A3	Inaccurate function visibility	**DISMISSED**

There are numerous cases where a function is declared `public` while it can be only externally called. Even though there are no array arguments that would result in gas savings, we recommend that the visibility be made accurate as good practice.
For example, in VaultBPool.sol and VaultCakePool.sol:

```
function pause() public
function setGov(address _govAddress) public
function setOnlyGov(bool _onlyGov) public
```

In BeltToken.sol

```
function claimInitialSupply(uint _amount) public
```

```
function getCirculatingSupply() public view returns (uint)
function mint(address _to, uint256 _amount) public onlyOwner
```

In MasterBelt.sol, MasterOrbit.sol

```
function set(uint256 _pid, uint256 _allocPoint, bool _withUpdate) public onlyOwner
function deposit(uint256 _pid, uint256 _wantAmt) public nonReentrant
function withdrawAll(uint256 _pid) public nonReentrant
function emergencyWithdraw(uint256 _pid) public nonReentrant
function inCaseTokensGetStuck(address _token, uint256 _amount) public onlyOwner
```

In StrategyAlpacaImpl.sol

```
function deposit(uint256 _wantAmt) public onlyOwner nonReentrant whenNotPaused
returns (uint256)
function wantLockedTotal() public view returns (uint256)
function setbuyBackRate(uint256 _buyBackRate, uint256 _buyBackPoolRate) public
function setGov(address _govAddress) public
function inCaseTokensGetStuck(address _token, uint256 _amount, address _to) public
```

In SingleSTrategyTokenImpl.sol

```
function supplyStrategy() public
function getPricePerFullShare() public view returns (uint)
function setBNBHelper(address _helper) public
```

Similarly in other strategy and strategyToken contracts.

A4	Declare immutable and constant variables	**DISMISSED**

For more clarity and gas savings, consider declaring all variables that are set during construction and could never be altered as immutables. Also variables that are statically assigned and never altered as constants.

In VaultBpool.sol (also VaultCakePool.sol)

```
// Dedaub: immutables
address public wantAddress;
address public masterBeltAddress;
```

In BeltToken.sol

```
// Dedaub: immutables
uint public initialSupply;
```

```
address public initialSupplyClaimer;
uint public startBlockMining;
// Dedaub: constants ( and underscores put by us, consider adopting for
readability)
uint public BELTPerBlock = 1_178_000_000_000_000_000;
address public burnAddress = 0x000000000000000000000000000000000000dEaD;
```

In MasterBelt.sol

```
// Dedaub: immutable. Also, no need to assign - vars are zero-initialized by
default
address public BELT = address(0);

// Dedaub: constants
address public burnAddress = 0x000000000000000000000000000000000000dEaD;
uint256 public ownerBELTReward = 178;
  // Dedaub: underscores put by us, consider adopting for readability
uint256 public BELTPerBlock = 1_000_000_000_000_000_000;
uint256 public startBlock = 5_559_747;
```

Similarly for MasterOrbit.sol but additionally:

```
// Dedaub: immutable.
address public orbitMinter;
```

In all of the earnV2/strategies/*Storage.sol and earnV2/strategiesV2/*Storage.sol contracts there are numerous variables that could be declared as immutable. For example, in StrategyAlpacaStorage.sol:

```
// Dedaub: immutables
address public uniRouterAddress;
bool public isWbnb;
address public vaultAddress;
address public wantAddress;
address public BELTAddress;
uint256 public poolId;
address[] public alpacaToWantPath;
address[] public alpacaToBELTPath;
address[] public wantToBELTPath;
```

In Single- and MultiStrategyTokenStorage.sol:

```
// Dedaub: immutable
bool public isWbnb;
```

A5	No-op now + 600 in swaps	**DISMISSED**

Swap calls to PancakeSwap add 600 to the current block's timestamp to compute the "deadline" for the swap order. For instance, the strategies contain code such as:

```
function earn() ... { ...
              IPancakeRouter02(uniRouterAddress).swapExactTokensForTokens(...
              now.add(600)
              );
}

function buyBack(uint256 _earnedAmt) … { ...
       IPancakeRouter02(uniRouterAddress).swapExactTokensForTokens(...
              now + 600
       );
  ...
}
```

Both forms of the addition are no-ops: the swap will always happen in the "current" block, from the perspective of the contract.

A6	Compiler known issues	**INFO**

The contracts were compiled with the Solidity compiler v0.6.12 which, at the time of writing, have some known bugs. We inspected the bugs listed for version 0.6.12 and concluded that the subject code is unaffected.

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.