Brokkr

Smart Contract Security Assessment

October 1, 2022

ABSTRACT

Dedaub was commissioned to perform a security audit of parts of the Brokkr protocol.

The Brokkr protocol is an investment protocol that uses automated techniques (portfolios) to diversify users by allocating deposits into multiple strategies and protocols with the goal to reduce the effects of market volatility. Brokkr manages to abstract away much of the complexity of creating a diversified DeFi portfolio and of setting up efficient strategies with features such as compounding of yield/rewards.

This audit report covers the contracts of the at the time private repository [block42-blockchain-company/bro-strategies] of the Brokkr protocol, up to commit hash 1957a32f91ae5c0ae2ad2f31002c04d9ac87904b.

The full audited contract list is the following:

```
contracts/
├─ common/
│   ├─ Common.sol
│   ├─ InvestmentToken.sol
│   │
│   ├─ bases/
│   │   ├─ FeeUpgradeable.sol
│   │   ├─ InvestmentLimitUpgradeable.sol
│   │   ├─ PortfolioBaseUpgradeable.sol
│   │   ├─ PortfolioOwnableBaseUpgradeable.sol
│   │   ├─ PortfolioOwnablePausableBaseUpgradeable.sol
│   │   ├─ StrategyBaseUpgradeable.sol
│   │   ├─ StrategyOwnableBaseUpgradeable.sol
│   │   └─ StrategyOwnablePausableBaseUpgradeable.sol
│   │
│   ├─ interfaces/*
│   │
│   ├─ libraries/
│   │   ├─ InvestableLib.sol
```

```
│   │   ├── InvestableLib_Test.sol
│   │   └── Math.sol
│   │
│   └── oracles/
│       ├── AaveOracle.sol
│       └── GmxOracle.sol
│
├── dependencies/*
│
├── portfolios/
│   └── percentageAllocation/
│       └── PercentageAllocation.sol
│
└── strategies/
    ├── cash/
    │   ├── Cash.sol
    │   └── CashStorageLib.sol
    │
    ├── stargate/
    │   ├── Stargate.sol
    │   └── StargateStorageLib.sol
    │
    ├── template/
    │   ├── Template.sol
    │   └── TemplateStorageLib.sol
    │
    └── traderjoe/
        ├── TraderJoe.sol
        └── TraderJoeStorageLib.sol
```

Following the audit Brokkr resolved a number of issues in this report and carried out fixes on the main branch of the repository, the net result of which can be see in commit 868bb961ade9f09788975a65b803ac644898d1d6. These were reviewed and accepted by the auditors. A number of other issues were acknowledged by Brokkr, as the team decided that no changes would be carried out in the present implementation. One issue was dismissed because it was mitigated by the way the Brokkr webapp's backend functions.

SETTING & 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.

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

Issue resolution includes "dismissed" or "acknowledged" but no action taken, by the client, or "resolved", per the auditors.

CRITICAL SEVERITY:

[No critical severity issues]

HIGH SEVERITY:

ID	Description	STATUS
H1	No minimum output amount is defined for calls to `SwapExactTokensForTokens()`	**RESOLVED**
The function `StrategyBaseUpgradeable::swapExactTokensForTokens()` swaps one token for another without specifying a minimum output amount. It is possible for attackers to tilt pools, resulting in swaps at disadvantageous rates for the strategies/portfolios. This affects both the `TraderJoe` and `Stargate` contracts; and can occur whenever USDC is swapped to a different token and back, and when Joe and Stg reward tokens are swapped for USDC.		
H2	Using `MasterchefJoe::addLiquidity` without defining minimum amounts for the pair might lead to loss of user funds	**RESOLVED**
Function `TraderJoe::_deposit()` calls the `addLiquidity` function of the MasterChefJoe contract providing a certain amount of tokenA and a corresponding amount of tokenB acquired by calling `swapExactTokensForTokens`. If we assume for simplicity that TraderJoe has 0 fees, swapping amountA of tokenA to tokenB will result in `amountB` being equal to `amountA * reserveB / (amountA + reserveA)`. After the swap the reserves will hold `newReserveA = reserveA + amountA` and `newReserveB = reserveB - amountB` tokens respectively. However, calling addLiquidity with amountA and amountB does not mean that both amounts will get deposited wholly. The pool expects us for amountA to deposit `desiredB`, where `desiredB = amountA * newReserveB / newReserveA` and as `newReserveB`		

< reserveB, desiredB will be less than amountB leading to amountB - desiredB not getting deposited to the pool. This amount of funds will remain in the strategy contract, as there is no contract logic to identify it and return it to the portfolio/user, but one should expect it to be small under normal conditions. However, if the TraderJoe pool has been tilted before the strategy's deposit one could expect the leftover amount to be of significant value, or in other words the bigger the tilt the bigger the amount of tokenA or tokenB that might not get deposited and become trapped in the strategy.

MEDIUM SEVERITY:

[No medium severity issues]

LOW SEVERITY:

ID	Description	STATUS
L1	The TraderJoe strategy assumes the price of the pair's stable coins to always be equal to 1 USD.	**DISMISSED**

The TraderJoe::getAssetValuations() function assumes that the price of the stable coins of the USDC-USDC.e pool in which it invests, is always equal to 1 USD. The valuation is computed by multiplying the percentage of the pool owned by the strategy with the total value of the pool's reserves.

```
// TraderJoe.sol::getAssetValuations():215
function getAssetValuations(bool, bool)
  public
  view
  virtual
  override
  returns (Valuation[] memory assetValuations)
```

```
{
  // ...

  assetValuations[0] = Valuation(
    address(strategyStorage.lpToken),
    (getTraderJoeLpBalance() * getTraderJoeLpReserve()) /
      strategyStorage.lpToken.totalSupply()
  );
}
```

The pool's value is returned by the function getTraderJoeLpReserve, as the sum of the USDC and the USDC.e reserves.

```
function getTraderJoeLpReserve() public view returns (uint256) {
  TraderJoeStorage storage strategyStorage = TraderJoeStorageLib
    .getStorage();

  (uint256 reserve0, uint256 reserve1, ) = strategyStorage
    .lpToken
    .getReserves();

  return reserve0 + reserve1;
}
```

This computation assumes that the value of the two aforementioned tokens is always equal to 1 USD. However, if the price drifts from this expected value, the getAssetValuations() function will start reporting a false valuation for the strategy and any portfolio using it.

***Comments*:**

The Brokkr team wished to clarify that the getAssetValuations() function reports prices in USDC. Hence, the assumption being made is not that USDC or USDC.e is always equal

to 1 USD, but that the price of USDC was always equal to the price of USDC.e. It was also mentioned that the reason for this assumption was that no reliable oracle for USDC.e could be found at the time of the audit.

L2	Minor loss of user funds when depositing to or withdrawing from Investables due to precision errors	**ACKNOWLEDGED**

After the function `PortfolioBaseUpgradeable::deposit` transfers the user's tokens to the contract, it splits and deposits them into the different investables according to the predefined allocation percentages.

```
// PortfolioBaseUpgradeable.sol::deposit():248
function deposit(
  uint256 amount,
  address investmentTokenReceiver,
  NameValuePair[] calldata params
) public virtual override nonReentrant {
  // ...
  depositToken.safeTransferFrom(_msgSender(), address(this), amount);
  // ...

  for (uint256 i = 0; i < investableDescs.length; i++) {
    // Dedaub: Potential precision errors due to integer division
    uint256 embeddedAmount = (amount *
      investableDescs[i].allocationPercentage) /
      Math.SHORT_FIXED_DECIMAL_FACTOR /
      100;
    if (embeddedAmount == 0) continue;
    depositToken.safeApprove(
      address(investableDescs[i].investable),
      embeddedAmount
    );
    investableDescs[i].investable.deposit(
      embeddedAmount,
      address(this),
```

```
        params
      );
    }
  }
```

The calculation of the `embeddedAmount` might suffer from precision errors due to integer division. This may result in depositing less funds in the underlying investables than what the user originally intended, while the difference (dust) gets trapped in the portfolio contract. Hence, when a user attempts to withdraw, they might end up receiving less than their initial deposit. In the worst case the dust amount will be less than `Math.SHORT_FIXED_DECIMAL_FACTOR * 100`, i.e., less than 10^5 wei. The calculation of withdrawal amounts in `PortfolioBaseUpgradeable::withdraw` could exhibit the same issue under certain amounts. However, it should be noted that the dust amounts due to the precision errors are expected to be considerably small in both cases (as already discussed), especially when the tokens used have 6 or even 18 decimals. Also, the issue can be largely mitigated by carefully choosing the allocation percentages, i.e., avoiding percentages with many decimals.

L3	Abstract contracts publicly expose owner only functionality	**ACKNOWLEDGED**

Several base abstract contracts like `PortfolioBaseUpgradeable` and `InvestmentLimitUpgradeable` implement owner only functionality without limiting its visibility or accessibility. For example, the `setInvestmentToken` function of the `PortfolioBaseUpgradeable` contract is defined as public and with no accessibility guards while it obviously implements functionality that should be used only by a limited set of entities. The abstract `PortfolioOwnableBaseUpgradeable` contract, which extends PortfolioBaseUpgradeable, "fixes" this by overriding the aforementioned function while adding the `onlyOwner` modifier.

```
function setTotalInvestmentLimit(uint256 totalInvestmentLimit)
  public
```

```
    virtual
    override
    onlyOwner
{
    super.setTotalInvestmentLimit(totalInvestmentLimit);
}
```

Nevertheless, the division of functionality and accessibility into different abstract contracts might lead to the introduction of bugs in future versions of the code. One way to avoid that would be to ensure that the critical functionality is never publicly accessible in any of the abstract contracts. In our example this could be achieved by removing `PortfolioBaseUpgradeable::setInvestmentToken` and introducing its internal counterpart `_setInvestmentToken`, which would be called by `PortfolioOwnableBaseUpgradeable::setInvestmentToken`.

[The issue has been acknowledged and partially resolved by the protocol team. According to the team, It will be fully addressed in a future release.]

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	`InvestmentToken` owner can mint tokens before transferring ownership	**ACKNOWLEDGED**
The `InvestmentToken` contract has a public onlyOwner mint() function, which would allow the owner of the contract to pre-mint portfolio or strategy tokens before transferring the ownership of the `InvestmentToken` contract to the respective portfolio or strategy. It is understood that ownership will be transferred as part of contract deployment to avoid such a situation.		

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	Babylonian square root method truncates result	**RESOLVED**

In `Math.sol`, the `sqrt` function truncates the decimal part of the answer due to integer division. For instance, the square root of 224 is 14.96, but sqrt(224) = 14. This can lead to a loss of precision in the answer of the `calculateMintAmount` function of the `InvestableLib` library.

```
// InvestableLib.sol::calculateMintAmount():32
function calculateMintAmount(
  TokenDesc[] memory tokenDescs,
  uint256 investableTokenSupply
)
```

Although the developers are most probably already aware of this, it is being mentioned for future reference, as the current contracts use another version of `calculateMintAmount` instead.

```
// InvestableLib.sol::calculateMintAmount():53
function calculateMintAmount(
  uint256 equitySoFar,
  uint256 amountInvestedNow,
  uint256 investmentTokenSupplySoFar
)
```

It could still prove beneficial to clearly document that the version of `calculateMintAmount`, which is mentioned first above, uses an approximation method for the calculation of the square root.

Comments:
The Brokkr team has removed calculateMintAmount(TokenDesc[] memory tokenDescs, uint256 investableTokenSupply) from the InvestibleLib library. As a result the sqrt() function in the Math library is no longer being called by any other function in the project.

A2	Claiming of fees cannot be achieved at the Portfolio level	**ACKNOWLEDGED**

In `PortfolioBaseUpgradable.sol`, the `claimFee()` function is empty and is not overridden anywhere else. To claim fees, the owner will have to call the `claimFee()` function on each individual strategy manually, since there is no facility to carry this out at the portfolio level.

A3	Several functions can be made private	**ACKNOWLEDGED**

The following functions could be declared private:
- `PortfolioBaseUpgradeable::findInvestableDescInd`
- `PortfolioBaseUpgradeable::deconstructNameValuePairArray`
- `PortfolioBaseUpgradeable::containsInvestableDesc`
- `PortfolioBaseUpgradeable::calculateEmbeddedFee`

A4	`__UUPSUpgradeable_init` is not always called	**RESOLVED**

The function `__UUPSUpgradeable_init` of `UUPSUpgradeable` is not called by the `initialize` functions of the following contracts that inherit `UUPSUpgradeable`:
- `PercentageAllocation`
- `Cash`
- `Stargate`
- `TraderJoe`

`__UUPSUpgradeable_init` may have an empty implementation at the moment but that could change in a future version of the Open Zeppelin code, which could be used

by an upgraded version of the aforementioned contracts. This omission of the call could lead to a serious bug If such a change went unnoticed or was not followed by the introduction of a call to `__UUPSUpgradeable_init`.		
A5	`investmentTokenReceiver` parameter might be 0	**RESOLVED**
Function `deposit` of `PortfolioBaseUpgradeable` and `StrategyBaseUpgradeable` defines the investmentTokenReceiver parameter, i.e., the address that will receive the investment token that corresponds to the deposit. There is no check to ensure that this address has not been accidentally set to 0.		
A6	`depositTokenReceiver` parameter might be 0	**RESOLVED**
Function withdraw of `PortfolioBaseUpgradeable` and `StrategyBaseUpgradeable` contracts does not guard against setting the parameter `depositTokenReceiver` to 0.		
A7	`feeReceiver_` parameter might be 0	**RESOLVED**
FeeUpgradeable::setFeeReceiver allows setting the receiver of the protocol fees to the 0 address, which could accidentally happen.		
A8	Empty functions and functions lacking implementation	**ACKNOWLEDGED**
The functions below are defined with an empty body. This may be related to future upcoming functionality which has not yet been implemented as of this audit. - `PortfolioBaseUpgradeable::claimFee` - `StrategyBaseUpgradeable::withdrawReward` - `Cash::_reapReward` - `Cash::getLiabilityBalances` - `Cash::getLiabilityValuations` - `Stargate::getLiabilityBalances` - `Stargate::getLiabilityValuations` - `TraderJoe::getLiabilityBalances` - `TraderJoe::getLiabilityValuations`		

A9	Compiler bugs	INFO
The code can be compiled with Solidity 0.8.0 or higher. For deployment, we recommend no floating pragmas, but a specific version, to be confident about the baseline guarantees offered by the compiler. Version 0.8.0, in particular, has some known bugs, which we do not believe 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, a public bug bounty program, as well as continuous security auditing and monitoring through Dedaub Watchdog.

ABOUT DEDAUB

Dedaub offers significant security expertise combined with cutting-edge program analysis technology to secure some of the most prominent protocols in DeFi. The founders, as well as many of Dedaub's auditors, have a strong academic research background together with a real-world hacker mentality to secure code. Protocol blockchain developers hire us for our foundational analysis tools and deep expertise in program analysis, reverse engineering, DeFi exploits, cryptography and financial mathematics.