Ebisu

Smart Contract Security Assessment

May 28, 2025

ABSTRACT

Dedaub was commissioned to perform a security audit of the Ebisu protocol, a friendly fork of Liquity V2. Among other things, the protocol's main additions include:

- Contract upgradeability
- Support for collateral tokens beyond WETH, stETH, and rETH
- The ability to dynamically add collateral tokens during the lifetime of the protocol

SETTING & CAVEATS

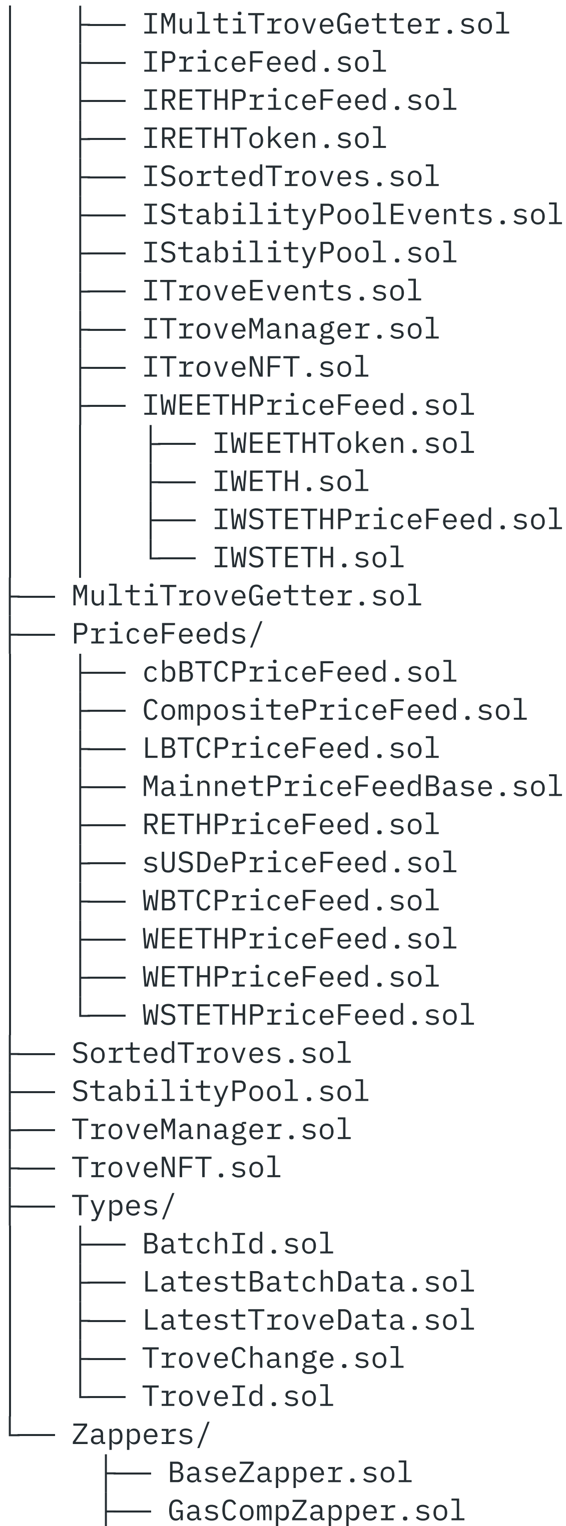
This audit report mainly covers the contracts of the **at-the-time private** repository https://github.com/ebisufinance/ebisu-money/ of the Ebisu Protocol at commit `dc6dc20cfb29994a8de5a9bd4d545d4acbec433f`.

Since the protocol is a friendly fork of LiquityV2, the team also reviewed all changes compared to LiquityV2's repository (https://github.com/liquity/bold) at commit `1652751c503a0a295a63b2c6b921d45d2f09a7fc`, which was assumed to be safe.

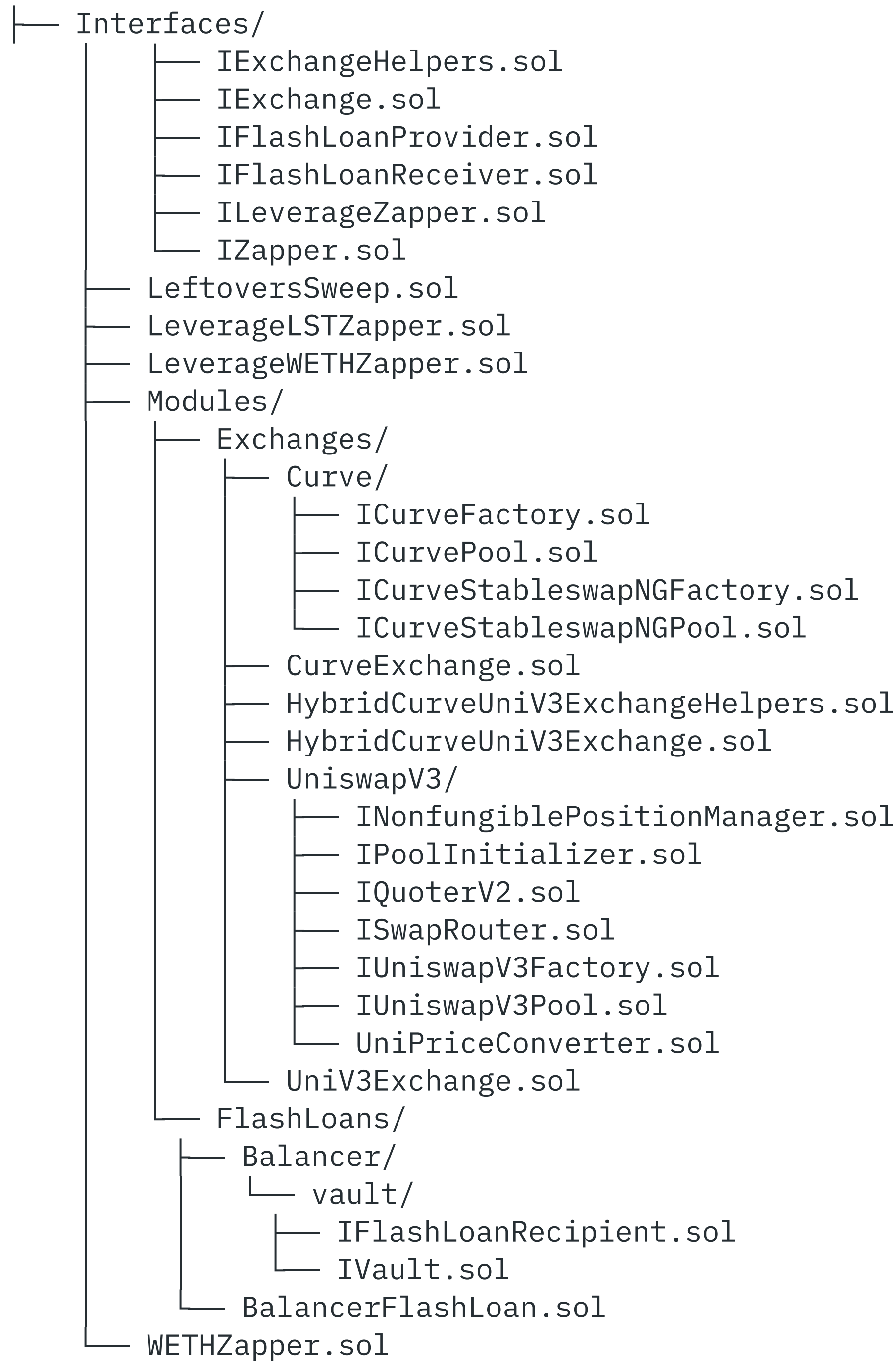
2 auditors worked on the codebase for **6** days on the following contracts:

```
contracts/src/
├── ActivePool.sol
├── AddressesRegistry.sol
├── BoldToken.sol
├── BorrowerOperations.sol
├── CollateralRegistry.sol
├── CollSurplusPool.sol
├── DefaultPool.sol
├── Dependencies/
│   ├── AddRemoveManagers.sol
│   ├── AggregatorV3Interface.sol
```

```
│       ├── BoStructs.sol
│       ├── Constants.sol
│       ├── DevStructs.sol
│       ├── EbisuUpgradeable.sol
│       ├── EbisuUpgradeableV2.sol
│       ├── LiquityBase.sol
│       ├── LiquityMath.sol
│       ├── Ownable.sol
│       ├── Structs.sol
│       └── TMStructs.sol
├── EbisuBorrowerOperationsHelper.sol
├── EbisuBranchFactory.sol
├── EbisuBranchManager.sol
├── GasPool.sol
├── HintHelpers.sol
├── Interfaces/
│       ├── IActivePool.sol
│       ├── IAddRemoveManagers.sol
│       ├── IAddressesRegistry.sol
│       ├── IBalancerRateProvider.sol
│       ├── IBoldRewardsReceiver.sol
│       ├── IBoldToken.sol
│       ├── IBorrowerOperations.sol
│       ├── ICollateralRegistry.sol
│       ├── ICollSurplusPool.sol
│       ├── ICommunityIssuance.sol
│       ├── IDefaultPool.sol
│       ├── IEbisuAdminRegistry.sol
│       ├── IEbisuBorrowerOperationsHelper.sol
│       ├── IEbisuBranchFactory.sol
│       ├── IEbisuBranchManager.sol
│       ├── IEbisuUpgradeable.sol
│       ├── IGovernance.sol
│       ├── IHintHelpers.sol
│       ├── IInterestRouter.sol
│       ├── ILBTCPriceFeed.sol
│       ├── ILiquityBase.sol
│       ├── ILQTYStaking.sol
│       ├── ILQTYToken.sol
│       ├── IMainnetPriceFeed.sol
```



```
│       ├── IMultiTroveGetter.sol
│       ├── IPriceFeed.sol
│       ├── IRETHPriceFeed.sol
│       ├── IRETHToken.sol
│       ├── ISortedTroves.sol
│       ├── IStabilityPoolEvents.sol
│       ├── IStabilityPool.sol
│       ├── ITroveEvents.sol
│       ├── ITroveManager.sol
│       ├── ITroveNFT.sol
│       ├── IWEETHPriceFeed.sol
│       │   ├── IWEETHToken.sol
│       │   ├── IWETH.sol
│       │   ├── IWSTETHPriceFeed.sol
│       │   └── IWSTETH.sol
├── MultiTroveGetter.sol
├── PriceFeeds/
│   ├── cbBTCPriceFeed.sol
│   ├── CompositePriceFeed.sol
│   ├── LBTCPriceFeed.sol
│   ├── MainnetPriceFeedBase.sol
│   ├── RETHPriceFeed.sol
│   ├── sUSDePriceFeed.sol
│   ├── WBTCPriceFeed.sol
│   ├── WEETHPriceFeed.sol
│   ├── WETHPriceFeed.sol
│   └── WSTETHPriceFeed.sol
├── SortedTroves.sol
├── StabilityPool.sol
├── TroveManager.sol
├── TroveNFT.sol
├── Types/
│   ├── BatchId.sol
│   ├── LatestBatchData.sol
│   ├── LatestTroveData.sol
│   ├── TroveChange.sol
│   └── TroveId.sol
└── Zappers/
     ├── BaseZapper.sol
     ├── GasCompZapper.sol
```



```
├── Interfaces/
│   ├── IExchangeHelpers.sol
│   ├── IExchange.sol
│   ├── IFlashLoanProvider.sol
│   ├── IFlashLoanReceiver.sol
│   ├── ILeverageZapper.sol
│   └── IZapper.sol
├── LeftoversSweep.sol
├── LeverageLSTZapper.sol
├── LeverageWETHZapper.sol
├── Modules/
│   ├── Exchanges/
│   │   ├── Curve/
│   │   │   ├── ICurveFactory.sol
│   │   │   ├── ICurvePool.sol
│   │   │   ├── ICurveStableswapNGFactory.sol
│   │   │   └── ICurveStableswapNGPool.sol
│   │   ├── CurveExchange.sol
│   │   ├── HybridCurveUniV3ExchangeHelpers.sol
│   │   ├── HybridCurveUniV3Exchange.sol
│   │   ├── UniswapV3/
│   │   │   ├── INonfungiblePositionManager.sol
│   │   │   ├── IPoolInitializer.sol
│   │   │   ├── IQuoterV2.sol
│   │   │   ├── ISwapRouter.sol
│   │   │   ├── IUniswapV3Factory.sol
│   │   │   ├── IUniswapV3Pool.sol
│   │   │   └── UniPriceConverter.sol
│   │   └── UniV3Exchange.sol
│   └── FlashLoans/
│       ├── Balancer/
│       │   └── vault/
│       │       ├── IFlashLoanRecipient.sol
│       │       └── IVault.sol
│       └── BalancerFlashLoan.sol
└── WETHZapper.sol
```

The audit's main target is security threats, i.e., what the community understanding would likely call "hacking", rather than the 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 and quantities returned from external protocols) is generally most effectively done through thorough testing rather than human auditing.

Because the protocol aims to support various Bitcoin wrapper assets (i.e., cbBTC, LBTC, WBTC), one of the audit's main focuses was ensuring that the accounting inherited from LiquityV2 was successfully adjusted to work with tokens that have 8 decimals of precision.

VULNERABILITIES & FUNCTIONAL ISSUES

This section details issues affecting 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:

[No high severity issues]

MEDIUM SEVERITY:

[No medium severity issues]

LOW SEVERITY:

ID	Description	STATUS
L1	Branch re-deployed without updating BranchManager	RESOLVED

Adding a new branch to Ebisu is achieved through a call to EbisuBranchFactory::_setupContractConnections, which in turn calls CollateralRegistry::addNewBranch for setup:

CollateralRegistry::addNewBranch:90

```
function addNewBranch(
        address _token,
        …,
        IEbisuBranchManager _branchManager
        ) external virtual onlyBranchFactory {
...
        if (collTokenToBranchManager[_token] == IEbisuBranchManager(address(0))) {
              collTokenToBranchManager[_token] = _branchManager;
        }
}
```

The issue is when a branch shuts down, a new branch may be redeployed for the same token. As can be seen in the above code, in this scenario the new branch manager will not be associated with the fresh branch, but the old branch manager will be. This is problematic, since the old branch manager will be pointing to old versions of the branch contracts.

ID	Description	STATUS
L2	Collateral gas compensation is effectively uncapped for 8-decimal assets	RESOLVED

During liquidations in which the debt is offset in the Stability Pool, a small portion of the collateral that is being seized is offered as compensation to the liquidator for the gas spent. In LiquityV2, this consists of 0.5% of the trove's liquidated with a cap at 2 ETH:

EbisuBranchManager::_getCollGasCompensation:889

```
function _getCollGasCompensation(uint256 _coll) internal pure returns (uint256) {
        return LiquityMath._min(_coll / COLL_GAS_COMPENSATION_DIVISOR,
COLL_GAS_COMPENSATION_CAP);
}
```

While COLL_GAS_COMPENSATION_DIVISOR being 200 represents 50/1000=0.5%, the protocol also has COLL_GAS_COMPENSATION_CAP be 2 ETH = 2e18 tokens.

For 8-decimal assets, this corresponds to 2e18 = 2e10 × 1e8 token units, ultimately yielding 2e10 tokens. This represents an unrealistically high amount—especially for assets wrapping BTC—so in practice, the collateral gas compensation is uncapped for those kinds of assets.

L3	Expression might round down to 0 for 8-decimal assets.	ACKNOWLEDGED

During liquidations that are offset in the Stability Pool, the corresponding percentage of the trove's collateral that will be offset is calculated based on the percentile of the trove's debt that is being offset as follows:

EbisuBranchManager::getOffsetAndRedistributionVals:347

```
...
collSPPortion = _entireTroveColl * debtToOffset / _entireTroveDebt;
...
```

When debtToOffset is a small enough number ($\epsilon << 10^{18}$) the above expression is approximately scaled at 8-18 < 0 decimals of precision, meaning that collSPPortion can end up being 0 when:

a. A liquidation takes place when the Stability Pool has been freshly created in a new 8-decimal collateral branch (i.e., SP_balance = ϵ)

[or]

b. A liquidation takes place when the Stability Pool has low BOLD balance just above 1e18 (i.e. SP_balance = $10^{18} + \epsilon$)

Since the minimum debt that can be created is 2000 USD (~0.02 BTC = 1e6 units of a BTC wrapper asset), if the protocol does not wish to directly address the potential scaling issue, it must ensure that the Stability Pool always holds more than 1e12 BOLD for offsets (in addition to the 1e18 minimum BOLD balance).

CENTRALIZATION CONSIDERATIONS:

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 considerations of this kind below. (These items 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	Some entities are considered partially trusted	**INFO**

The ebisuAdmin is an address that:

- Can change important branch parameters in EbisuBranchManager
- Can control who has access to proposing parameters in EbisuBranchManager
- Can initiate an upgrade proposal for any protocol contract
- Can set the factory responsible for creating a new collateral branch
- Can add support for more collateral tokens

Although this entity has critical privileges over the protocol, parameters cannot be arbitrarily set but instead are typically after certain sanity checks take place.

Additionally, the version of upgradeable contracts that are present in the protocol cannot be upgraded atomically. Instead, each upgrade can only take effect after a timelock period passes (7 days).

More importantly, the protocol aims at giving all those privileges to governance in the future, preventing a single address from having the above-mentioned privileges.

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	Different constraints on CCR	**INFO**
	`AddressRegistry` expects `_ccr` to be between `1e18` and `2e18` (line 83) however `EbisuBranchManager` line 162 expects `_CCR` to be between `MCR` and `3e18`.	
A2	Unreachable revert	**INFO**
	`EbisuBranchManager::setMinAnnualInterestRate` on line 176 reverts if `rate <= 0`, however the previous line reverts when `_rate <= MIN_ANNUAL_INTEREST_RATE`. `MIN_ANNUAL_INTEREST_RATE` is a `uint256`, thus when `_rate <= 0` this previous revert will trigger.	
A3	Redundant permissions	**INFO**
	The following functions give redundant calling permissions (i.e., the implementation of the contract given permission never actually calls the function): 1. `DefaultPool::sendCollToActivePool` to `EbisuBranchManager`. 2. `CollSurplusPool::accountSurplus` to `EbisuBranchManager`.	

3. ActivePool::setShutdownFlag to TroveManager.
4. SortedTroves::remove to BorrowerOperations.
5. SortedTroves::insert to BorrowerOperationsHelper.
6. SortedTroves::reInsert to BorrowerOperations.
7. SortedTroves::reInsertBatch to BorrowerOperations.

A4	Redundant assignments	INFO

1. EbisuBranchManager lines 113, 115, and 118 are redundant; these set activePool, priceFeed, and defaultPool which are already set by the call to LiquityBase::initialize on line 109.
2. EbisuBranchManager line 130 sets debtCap = 0, which is unnecessary since this is the default value.
3. TroveManager line 747 sets Troves[_troveId].lastInterestRateAdjTime to the latest timestamp. This is unneeded since it is already performed by the call to _initializeTrove on line 744.

A5	Redundant finalisation of never-proposed parameter	INFO

EbisuBranchManager::finalizeParameterChange finalises proposals for changes to several parameters, including for debtCap. For the latter this is not needed, since debtCap is set immediately by EbisuBranchManager::setDebtCap on line 204, and is never added as a proposal to proposedValues.

A6	Mismatch between comment and implementation	INFO

The comment on line EbisuBranchManager:301 states that only ebisuAdmin should call EbisuBranchManager::setMinDebt, but it uses the modifier onlyAuthorized(Parameter.MIN_DEBT) rather than onlyEbisuAdmin.

A7	Unused internal functions	INFO

The following internal functions are not used anywhere:

- `EbisuBranchFactory::getBytecode`
- `EbisuBranchFactory::getAddress`
- `TroveManager::_getCollGasCompensation`

A8	Users can theoretically create a profitable arbitrage if sUSDe's USDe backing is reduced	**INFO**

One of the collateral assets to be supported is Ethena's sUSDe. sUSDe is an ERC-4626 vault implementation for Ethena's stable coin - USDe - which is meant to tokenize the yield that the protocol generates from various sources.

The asset's price feed somewhat resembles LiquityV2's logic for RETH:
*if RETH-ETH and RETH-ETH_exrate within 2%: max(RETH-ETH, RETH-ETH_exrate) * ETH-USD else: min(RETH-ETH, RETH-ETH_exrate) * ETH-USD*

`sUSDePriceFeed::_fetchPricePrimary:65`

```
...
// Get canonical rate using previewRedeem
uint256 sUsdeRate = sUsde.previewRedeem(1 ether);
uint256 canonicalPrice = usdeUsdPrice * sUsdeRate / 1e18;

uint256 price;
// If it's a redemption and canonical is within 2% of market, use the max to
mitigate unwanted redemption oracle arb
if (_isRedemption && _withinDeviationThreshold(sUsdeUsdPrice, canonicalPrice,
SUSDE_USD_DEVIATION_THRESHOLD)) {
       price = LiquityMath._max(sUsdeUsdPrice, canonicalPrice);
} else {
       // Take the minimum of (market, canonical) in order to mitigate against
upward market price manipulation
       price = LiquityMath._min(sUsdeUsdPrice, canonicalPrice);
}
...
```

However, sUSDe being a typical ERC-4626 implementation means that sUSDe's exchange rate is given as:

0x9d39a5de30e57443bff2a8307a4256c8797a3497::ERC4626::_convertToAssets:207

```
function totalAssets() public view virtual override returns (uint256) {
    return _asset.balanceOf(address(this));
}

...
function _convertToAssets(uint256 shares, Math.Rounding rounding) internal view
virtual returns (uint256) {
      return shares.mulDiv(totalAssets() + 1, totalSupply() + 10 **
_decimalsOffset(), rounding);
}
...
```

It becomes apparent that the asset's rate can be inflated by directly transferring USDe to the sUSDe contract–unlike RETH's rate, which is strictly calculated based on Rocket Pool's internal accounting.

Since the oracle always returns the minimum price between Chainlink's sUSDe feed and sUSDe's canonical rate, a user receives the best possible price when a redemption occurs and the two prices differ by more than the set deviation threshold (2%). When the collateral is priced lower, the same amount of BOLD tokens corresponds to more collateral tokens.

A possible scenario would involve a user:

1. Inflating the sUSDe exchange rate (atomically)
2. Redeeming a large amount of BOLD tokens
3. With the redemption occurring at the market price, the user then redeems all received sUSDe for USDe at the inflated rate

This represents a buy-low, sell-high scheme—constituting a simple arbitrage loop. A thorough risk analysis of this scenario can be found here: https://hackmd.io/@DpO5w3VJTCC5KnEfWwp07Q/SJhT8Mzzel

Ultimately, as long as sUSDe's USDe balance is in the billions, there is no practical risk of an attack using this method. Even when the balance falls to the millions, only users redeeming millions of BOLD would be in a position to execute this arbitrage loop.

A9	Compiler bugs	INFO

The code is compiled with Solidity `0.8.24`. Version `0.8.24` has no known bugs.

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 Security Suite.

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.