Gravita Protocol

Security Assessment

Apr 3, 2023

ABSTRACT

Dedaub was commissioned to perform a security audit of the Gravita protocol, an adaptation of the Liquity codebase to a multi-collateral system. Gravita follows the example of other Liquity-inspired codebases, such as Vesta and Yeti, but introduces its own design choices. Importantly, the protocol:

- allows multiple kinds of collateral (“assets”);
- strictly segregates the different collateral kinds, with respect to liquidation/redemption behavior. Borrowing positions (“vessels”) are strictly associated with one asset, and the financials of any one asset should not affect in any way the others.

The audit was performed at commit hash `bfa97cb37dc0e2927e75b36753585140f25a26dd` of the gravita-SmartContracts repository. Updates were inspected based on commit deltas, as listed, considering only whether the change addresses the issue at hand, and not in terms of considering the overall codebase.

SETTING & CAVEATS

An audit team of two auditors worked on the codebase for 2 working weeks.

The audit scope was defined in coordination with the client and consists of the following files:

```
AdminContract.sol
BorrowerOperations.sol
DebtToken.sol
FeeCollector.sol
PriceFeed.sol
StabilityPool.sol
VesselManager.sol
VesselManagerOperations.sol
```

Other files in the codebase are considered trusted, although we consulted some of them during the audit, e.g., to inspect token transfers in `ActivePool.sol`, specifically in regards to reentrancy threats.

The above in-scope files comprise around 4885 lines of code. This size extends well beyond what can be audited from scratch in the allotted time. However, the audit was conducted as a delta audit, assuming that the original Liquity codebase is well-trusted. We, therefore, audited the changes relative to the Liquity repository, commit hash `26ff1ce48b00d7632c459a9176574133b955b7be`, dated July 20, 2022, i.e., the stable Liquity version around the time of origination of the Gravita project codebase.

Given that the Liquity codebase is well-trusted, the delta audit approach allowed us to get high confidence in a much more realistic timeframe. However, it comes with caveats, even beyond the correctness of the Liquity code itself. Notably, there is a need to maintain the Liquity invariants throughout, including in files outside the audit scope. (E.g., a single quantity with the wrong number of decimals can violate correctness.) In our audit, we found the code to reflect a good understanding of Liquity, so we have reasonable confidence that Liquity invariants are preserved.

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. Although a high-level specification describing interactions within the protocol was provided, functional correctness (i.e. issues in "regular use") was a secondary consideration. 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.

PROTOCOL-LEVEL CONSIDERATIONS

There are several important protocol-level considerations that the development team should be well aware of.

- There is a REDEMPTION_SOFTENING_PARAM set to 97%. This has the effect that third-party (not the vessel's owner) redeemers only receive 97% of the collateral. As a result, the GRAI token will likely not be well-pegged to the USD, but more lower.
- Unlike in Liquity, the `baseRate` quantity is used to adjust the fees only for redemptions and NOT for borrowing. (In Liquity, if there are recent redemptions, there are higher fees both for redemptions and for borrowing: https://github.com/liquity/dev#intuition-behind-fees)

  In Gravita, the `baseRate` is per-asset, and, thus, only affects redemptions. The cost of borrowing is entirely unaffected by whether there have been recent redemptions: the formula for borrowing fees just includes the static per-asset fee, not the dynamically-changing `baseRate`.

  This is a fine design choice, but it is clearly a matter of financial protocol design.
- The Gravita fees calculation permits limited economic gaming. For instance, fee-wise it may be preferable for a client to open a new debt position instead of increasing his/her old one, just because of the fee refund curve, which remains linear.

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 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:

ID	Description	STATUS

M1	Reentrancy considerations	**ACKNOWLEDGED + RESOLVED** (d8bce016)

There are some reentrancy considerations throughout the codebase. The code has apparently not been developed with reentrancy in mind (i.e., with no consideration that transferring tokens may yield control to an untrusted party). It is not wise to rely on a security audit to find all possible sources of reentrancy, if developers have not first thought deeply about them. Therefore, we recommend the use of the protocol only with tokens that do not have callbacks to untrusted parties, to completely mitigate all reentrancy concerns.

However, we list below what we believe are all the reentrancy threats, viewed conservatively. Our suggestions aim to bring the codebase to the same levels of reentrancy safety as the Liquity codebase.

- The protocol should not be used with collateral tokens that perform callbacks to the **sender** of tokens. E.g., ERC 777 tokens have this behavior and they are well-known as reentrancy death traps. Since these tokens are very rare, this is not a true limitation.

  The specific threat with such tokens is that the internal function BorrowerOperations::_activePoolAddColl (which does a `transferFrom` from an untrusted party) is used in the middle of code that has complex effects after the function call. Viewed differently, such tokens are very different from ETH in callback behavior, and since the Liquity codebase of origin has been written and tested with ETH in mind, such unintuitive callbacks can yield major violations of the checks-effects-interactions pattern.

```
function _activePoolAddColl(
        address _asset,
        IActivePool _activePool,
        uint256 _amount
) internal {
        IERC20Upgradeable(_asset).safeTransferFrom(
                msg.sender,
                address(_activePool),
                SafetyTransfer.decimalsCorrection(_asset, _amount)
        );
        _activePool.receivedERC20(_asset, _amount);
}

// called in:

function openVessel(
        address _asset,
        uint256 _assetAmount,
        uint256 _debtTokenAmount,
        address _upperHint,
        address _lowerHint
) external override {
   …
   _activePoolAddColl(vars.asset, contractsCache.activePool, _assetAmount);
   _withdrawDebtTokens(
                vars.asset,
                contractsCache.activePool,
                contractsCache.debtToken,
                msg.sender,
                _debtTokenAmount,
                vars.netDebt
  );
  // Move the debtToken gas compensation to the Gas Pool
  _withdrawDebtTokens(
                vars.asset,
                contractsCache.activePool,
                contractsCache.debtToken,
                gasPoolAddress,
                adminContract.getDebtTokenGasCompensation(vars.asset),
                adminContract.getDebtTokenGasCompensation(vars.asset)
  );
  …
}
```

- Compared to Liquity, there are places in the code where effects take place after external interactions, i.e., after sending tokens to an untrusted recipient. We recommend reordering rather than trying to reason that all these interactions are reentrancy-safe.

- In BorrowerOperations::closeVessel:

```
// Send the collateral back to the user
activePoolCached.sendAsset(_asset, msg.sender, coll);
// Signal to the fee collector that debt has been paid in full
feeCollector.closeDebt(msg.sender, _asset);
```

- In BorrowerOperations::_activePoolAddColl (also discussed in previous item):

```
function _activePoolAddColl(
        address _asset,
        IActivePool _activePool,
        uint256 _amount
) internal {
        IERC20Upgradeable(_asset).safeTransferFrom(
              msg.sender,
              address(_activePool),
              SafetyTransfer.decimalsCorrection(_asset, _amount)
        );  // Dedaub: reverse transfer and next call?
        _activePool.receivedERC20(_asset, _amount);
}
```

- In StabilityPool::_sendGainsToDepositor, there are effects after a transfer, but these seem safe: some of the effects are transfers (inside the loop) over entirely different tokens, some others are just a subtraction of collaterals, which should be fine if deferred. However, we recommend reordering the transfers and `totalColl.amounts` adjustment, out of an abundance of caution.

```
function _sendGainsToDepositor(
        address _to,
        address[] memory assets,
        uint256[] memory amounts
    ) internal {
  …
  for (uint256 i = 0; i < assetsLen; ++i) {
    …
    IERC20Upgradeable(asset).safeTransferFrom(address(this), _to,
                                                  amount);
  } // Dedaub: effects after call.
  totalColl.amounts = _leftSubColls(totalColl, assets, amounts);

  …
}
```

LOW SEVERITY:

ID	Description	STATUS
L1	PriceFeed::addOracle does not check if it is overwriting a previous queued oracle	**RESOLVED (not applicable as of e4fbfc30)**

In PriceFeed::addOracle, the `queuedOracles` entry for the token is written without checking whether it is zero. This is only a problem in case the controller makes a mistake, but the presence of a `deleteQueuedOracle` function suggests that the right behavior for a controller would be to delete a queued oracle if it's no longer valid.

```
function addOracle(address _token, address _chainlinkOracle, bool _isEthIndexed)
external override isController {
  AggregatorV3Interface newOracle = AggregatorV3Interface(_chainlinkOracle);
  _validateFeedResponse(newOracle);
  if (registeredOracles[_token].exists) {
    uint256 timelockRelease = block.timestamp.add(_getOracleUpdateTimelock());
    queuedOracles[_token] = OracleRecord(newOracle, timelockRelease, true,
                                         true, _isEthIndexed);
  } else {
    registeredOracles[_token] = OracleRecord(newOracle, block.timestamp, true,
                                         true, _isEthIndexed);
    emit NewOracleRegistered(_token, _chainlinkOracle, _isEthIndexed);
  }
}
```

```
function deleteQueuedOracle(address _token) external override isController {
  delete queuedOracles[_token];
}
```

L2	The timelock for adding oracles can be circumvented by deleting the previous oracle	**RESOLVED (not applicable as of e4fbfc30)**

On the same code as issue L1, in the PriceFeed contract, the controller can always subvert the above timelock by just deleting the registered oracle.

```
function deleteOracle(address _token) external override isController {
  delete registeredOracles[_token];
}
```

Thus, the timelock can only prevent accidents in the controller, and not provide assurances of having a delay for review of changes to oracles.

L3	A series of liquidations can cause the zeroing of `totalStakes`	**ACKNOWLEDGED**

The stake of a Vessel holding `_asset` as collateral is computed by the formula in VesselManager::_computeNewStake :

```
stake =
_coll.mul(totalStakesSnapshot[_asset]).div(totalCollateralSnapshot[_asset]);
```

The stake is updated when the Vessel is adjusted and `_coll` is the new collateral amount of the Vessel and `totalStakesSnapshot`, `totalCollateralSnapshot` the total stakes and total collateral respectively right after the last liquidation.

A liquidation followed by a redistribution of the debt and collateral to the other Vessels decreases the total stakes (the stake of the liquidated Vessel is just deleted and not shared among the others) and the total collateral (if we ignore the fees) does not change. Therefore the ratio in the above formula is constantly decreasing after each liquidation followed by redistribution and each new Vessel will get a relatively smaller

stake. The finite precision of the arithmetic operations can lead to a zeroing of `totalStakes`, if a series of liquidations of Vessels with high stakes occurs. If this happens, the total stakes will be zero forever and each new vessel will be assigned a zero stake.
If this happens many functionalities of the protocol are blocked i.e. the VesselManager::redistributeDebtAndCollateral will revert every time, since the debt and collateral to distribute are computed dividing by the (zero) totalStakes.
The probability of such a problem is higher in Gravita, compared to Liquity, because Gravita allows multiple collateral assets, some of them, in principle, more volatile compared to ETH.

L4	PriceFeed::fetchPrice could return arbitrarily stale prices, if Chainlink Oracle's response is not valid	**RESOLVED (e4fbfc30)**

The protocol uses the PriceFeed::fetchPrice to get the price of a `_token`, whenever it needs to. This function first calls the Chainlink oracle to get the price for this `_token` and then checks the validity of the response. If it is valid, it stores the answer in `lastGoodPrice[_token]` and also returns it to the caller. If the Chainlink response is not valid, then the function returns the value stored in `lastGoodPrice[_token]`. The problem is that this value could have been stored a long time ago and there is no check about this in the contract. As an edge case, if the Chainlink oracle does not give a valid answer, upon its first call for a `_token`, then the PriceFeed::fetchPrice function will return a zero price. Liquity uses a secondary oracle, if the response of Chainlink is not valid, and only if both oracles fail, the stored last good price is being used, but in Gravita there is no secondary oracle.

L5	AdminContract::sanitizeParameters has no access control	**RESOLVED (58a41195)**

The function sets important collateral data (to default values) yet has no access control, unlike, e.g., the almost-equivalent `setAsDefault`, which is `onlyOwner`.

Although there are many other safeguards that ensure that collateral is valid, we recommend tightening the access control for `sanitizeParameters` as well.

```
function sanitizeParameters(address _collateral) external {
	if (!collateralParams[_collateral].hasCollateralConfigured) {
		_setAsDefault(_collateral);
	}
}
function setAsDefault(address _collateral) external onlyOwner {
	_setAsDefault(_collateral);
}
```

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	Whitelisted contracts can mint arbitrarily large amounts of debt tokens	**INFO** **(acknowledged)**

The role of the whitelisted contracts is not completely clear to us. There is only one related comment in `DebtToken.sol`:

```
// stores SC addresses that are allowed to mint/burn the token (AMO strategies, L2 suppliers)
mapping(address => bool) public whitelistedContracts;
```

These contracts can mint debt tokens without depositing any collateral calling DebtToken::mintFromWhitelistedContract. This could be a serious problem if such a contract was malicious. Also, even if these contracts work as expected, minting debt tokens without providing any collateral could have a serious impact on the price of the debt token.		
N2	Protocol owners can set crucial parameters	**INFO** **(acknowledged)**
Key functionality is trusted to the owner of various contracts. Owners can set the kinds of collateral accepted, the oracles that are used to price collateral, etc. Thus, protocol owners should be trusted by users.		

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	In struct `Vessel` (IVesselManager.sol), `asset` is unnecessary	**INFO**
Field `asset` of struct `Vessel` is currently unused. `Vessel` records are currently only used in a mapping that has the asset as the key, so there is no need to read the asset from the Vessel data.		
A2	In FeeCollector::_decreaseDebt no need to check for refundable fees if the expiration time of the refunding is `block.timestamp`	**INFO**

In the code below

```
if (mRecord.to < NOW) {
                        _closeExpiredOrLiquidatedFeeRecord(_borrower, _asset, mRecord.amount);
                }
```

< can be replaced by <=, since when `mRecord == NOW`, there is nothing left for the user to refund.

A3 Unused event — INFO

The following event is declared in `IAdminContract.sol` but not used anywhere:

```
event MaxBorrowingFeeChanged(uint256 oldMaxBorrowingFee, uint256 newMaxBorrowingFee);
```

A4 Unused storage variables — INFO

The storage mapping StabilityPool::pendingCollGains and code accessing it are unnecessary since the information is never set to non-zero values.

```
// Mapping from user address => pending collaterals to claim still
// Must always be sorted by whitelist to keep leftSumColls functionality
mapping(address => Colls) pendingCollGains;
...
function getDepositorGains(address _depositor) public view
 returns (address[] memory, uint256[] memory) {
        …
        // Add pending gains to the current gains
        return (
               collateralsFromNewGains,
               _leftSumColls(
                      Colls(collateralsFromNewGains, amountsFromNewGains),
                      pendingCollGains[_depositor].tokens,
                      pendingCollGains[_depositor].amounts
               )
        );
}
...
function _sendGainsToDepositor(
        address _to,
        address[] memory assets,
        uint256[] memory amounts
) internal {
    ...
    // Reset pendingCollGains since those were all sent to the borrower
    Colls memory tempPendingCollGains;
    pendingCollGains[_to] = tempPendingCollGains;
}
```

Also, StabilityPool::controller is unused and never set:

```
IAdminContract public controller;
```

Finally, variables `activePool`, `defaultPool` in GravitaBase seem unused and not set (at least for most subcontracts of GravitaBase).

```
IActivePool public activePool;
IDefaultPool internal defaultPool;
```

A5 TransferFrom is really just a transfer — INFO

In StabilityPool::_sendGainsToDepositor, it is not clear why the `transferFrom` is not merely a `transfer`.

```
function _sendGainsToDepositor(
        address _to,
        address[] memory assets,
        uint256[] memory amounts
    ) internal {
  …
  for (uint256 i = 0; i < assetsLen; ++i) {
    …
    IERC20Upgradeable(asset).safeTransferFrom(address(this), _to,
                                              amount);
  }
  …
}
```

A6	Tokens with more than 18 decimals are not supported	INFO

Tokens with more than 18 decimals are not supported, based on the SafetyTransfer library (outside the audit scope).

```
function decimalsCorrection(address _token, uint256 _amount)
       internal
       view
       returns (uint256)
   {
       if (_token == address(0)) return _amount;
       if (_amount == 0) return 0;

       uint8 decimals = ERC20Decimals(_token).decimals();
       if (decimals < 18) {
             return _amount.div(10**(18 - decimals));
       }

       return _amount; // Dedaub: more than 18 not supported correctly!
   }
```

We do not recommend trying to address this, as it may introduce other complexities for very little practical benefit. Instead, we recommend just being aware of the limitation.

A7	No-op statement (consisting of a mere expression)	INFO

In BorrowingOperations::openVessel, the following expression (used as a statement!) is a no-op:

```
vars.debtTokenFee;
```

A8	Unused external function, not called as expected	INFO

BorrowerOperations::moveLiquidatedAssetToVessel appears to not be used in the protocol.

```
    // Send collateral to a vessel. Called by only the Stability Pool.
    function moveLiquidatedAssetToVessel(
        address _asset,
        uint256 _amountMoved,
        address _borrower,
        address _upperHint,
        address _lowerHint
    ) external override {
        _requireCallerIsStabilityPool();
        _adjustVessel(_asset, _amountMoved, _borrower, 0, 0, false,
                       _upperHint, _lowerHint);
    }
```

A9	Unnecessary `isInitialized` flags	**INFO**

The following pattern over storage variable `isInitialized` appears in several contracts but should be entirely unnecessary, due to the presence of the `initializer` modifier.

```
bool public isInitialized;

function setAddresses(...) external initializer {
        require(!isInitialized);
        …
        isInitialized = true;
}
```

Contracts with the pattern include FeeCollector, PriceFeed, ActivePool, CollSurplusPool, DefaultPool, SortedVessels, StabilityPool, VesselManager, VesselManagerOperations, CommunityIssuance, GRVTStaking.

A10	Anachronisms	**INFO**

The codebase exhibits some old code patterns (which we do not recommend fixing, since they directly mimick the Liquity trusted code):

- The use of `assert` for condition checking (instead of `require/if→revert`). (Some of the asserts have been replaced, but not all.)
- The use of SafeMath instead of relying on Solidity 0.8.* checks.

A11	Unnecessary and error-prone use of `this.*`	**INFO**

Some same-contract function calls are made with the pattern this.func(), which causes a new internal transaction and changes the msg.sender. This should be avoided for clarity and (gas) performance.

In VesselManager:

```
function isVesselActive(address _asset, address _borrower) public view override
returns (bool) {
        return this.getVesselStatus(_asset, _borrower) == uint256(Status.active);
}
```

In PriceFeed (and also note the unusual convention of 0 = ETH):

```
function _calcEthPrice(uint256 ethAmount) internal returns (uint256) {
        uint256 ethPrice = this.fetchPrice(address(0));
         // Dedaub: Also, why the convention that 0 = ETH?
        return ethPrice.mul(ethAmount).div(1 ether);
}
…
function _fetchNativeWstETHPrice() internal returns (uint256 price) {
        uint256 wstEthToStEthValue = _getWstETH_StETHValue();
        OracleRecord storage stEth_UsdOracle = registeredOracles[stethToken];
        price = stEth_UsdOracle.exists ? this.fetchPrice(stethToken) :
                                          _calcEthPrice(wstEthToStEthValue);
        _storePrice(wstethToken, price);
}
```

A12	Compatibility of PriceFeed::_fetchPrevFeedResponse, _isValidResponse with future versions of the Chainlink Aggregator	**INFO**

The `roundId` returned by the Chainlink AggregatorProxy contract is a `uint80`. The 16 most important bits keep the `phaseId` (incremented every time the underlying aggregator is updated) and the other 64 bits keep the `roundId` of the aggregator. As long as the underlying aggregator is the same, the `roundId` returned by the proxy will increase by one in each new round, but in an update of the aggregator contract the proxy `roundId` will increment not by 1, since the `phaseId` will also change. In this case the previous round is not current_roundId-1 and `_fetchPrevFeedResponse` will not return the price data from the previous round (which was a round of the previous aggregator). We mention this issue, although the probability that the protocol fetches a price at the time of an update of a Chainlink oracle is relatively small and each round lasts a few minutes to an hour.

`PriceFeed::_isValidResponse` does all the validity checks necessary for the current [Chainlink Aggregator version](). Chaninlink's AggregatorProxy::latestRoundData returns also two extra values `uint256 startedAt`, `uint80 answeredInRound`, which, for the current version, do not hold extra information i.e. `answeredInRound==roundId`, but in past and possible future versions they could be used for some extra validity checks i.e. `answeredInRound>=roundId`.

A13	Unnecessary code	**INFO**

In BorrowerOperations:

```
function _requireNonZeroAdjustment(
        uint256 _collWithdrawal,
        uint256 _debtTokenChange,
        uint256 _assetSent
    ) internal view {
        require(
              msg.value != 0 || _collWithdrawal != 0 || _debtTokenChange != 0 ||
              _assetSent != 0,    // Dedaub: `msg.value != 0` not possible
              "BorrowerOps: There must be either a collateral change or a debt
change"
       );
}
```

the condition `msg.value != 0` is not possible, as ensured in the single place where this function is called (`_adjustVessel`). The condition should be kept if the function is to be usable elsewhere in the future.

Similarly, in VesselManager, the condition marked with a comment below seems unnecessary, given that the arithmetic is compiler-checked.

```
function decreaseVesselDebt(
       address _asset,
       address _borrower,
       uint256 _debtDecrease
    ) external override onlyBorrowerOperations returns (uint256) {
       uint256 oldDebt = Vessels[_borrower][_asset].debt;
       if (_debtDecrease == 0) {
             return oldDebt; // no changes
       }
       uint256 paybackFraction = (_debtDecrease * 1 ether) / oldDebt;
       uint256 newDebt = oldDebt - _debtDecrease;
       Vessels[_borrower][_asset].debt = newDebt;
       if (paybackFraction > 0) {
             if (paybackFraction > 1 ether) {
             // Dedaub:Impossible. The "-" would have reverted, three lines above
                   paybackFraction = 1 ether;
             }
             feeCollector.decreaseDebt(_borrower, _asset,
                                        paybackFraction);
       }
       return newDebt;
}
```

A14	Unclear ownable policy	**INFO**

Some contracts are defined to be Ownable (using the OZ libraries), yet do not use this capability (beyond initialization). These include:

- StabilityPool initializes Ownable, relinquishes ownership, but never checks ownership in `setAddresses`, or elsewhere.

```
function setAddresses(
        address _borrowerOperationsAddress,
        address _vesselManagerAddress,
        address _activePoolAddress,
        address _debtTokenAddress,
        address _sortedVesselsAddress,
        address _communityIssuanceAddress,
        address _adminContractAddress
    ) external initializer override {
    …
        __Ownable_init();
    …
        renounceOwnership();
    // Dedaub: The function was onlyOwner in Liquity, here there's
    //  no point of Ownable
}
```

- VesselManagerOperations inherits and initializes ownable functionality but is it used?

```
function setAddresses(
        address _vesselManagerAddress,
        address _sortedVesselsAddress,
        address _stabilityPoolAddress,
        address _collSurplusPoolAddress,
        address _debtTokenAddress,
        address _adminContractAddress
    ) external initializer {
    …
        __Ownable_init();  // YS:! why?
}
```

A15	No explicit check in BorrowerOperations::openVessel that the collateral deposited by the user is approved	**INFO**

If a user attempts to open a Vessel with a collateral asset not approved by the owner, the transaction will fail, because there will be no price oracle registered for this asset. Therefore it is checked if the user deposits an approved collateral asset, but only indirectly. It would be better if there was an explicit check.

A16	AdminContract::addNewCollateral only partially initializes the `collateralParams` structure	**INFO**

We cannot find a specific problem with the current only partial initialization, since even if the owner just adds a new `_collateral` and does not set all the fields of `collateralParams[_collateral]`, upon opening a Vessel the protocol sets the default values for these. But, in general it is not a good practice to leave uninitialized variables and it would be better if in `addnewCollateral` the owner also set the default values for the remaining `collateralParams` elements.

A17	Unused internal functions	**INFO**

In StabilityPool, the following two functions are unused.

```
function _requireUserHasVessel(address _depositor) internal view {
        address[] memory assets = adminContract.getValidCollateral();
        uint256 assetsLen = assets.length;
        for (uint256 i; i < assetsLen; ++i) {
              if (vesselManager.getVesselStatus(assets[i], _depositor) == 1) {
                    return;
              }
        }
        revert("StabilityPool: caller must have an active vessel to withdraw
AssetGain to");
}

function _requireUserHasAssetGain(address _depositor) internal view {
        (address[] memory assets, uint256[] memory amounts) =
          getDepositorGains(_depositor);
        for (uint256 i = 0; i < assets.length; ++i) {
               if (amounts[i] > 0) {
                      return;
               }
        }
        revert("StabilityPool: caller must have non-zero gains");
}
```

A18	Misspellings/content mistakes in names or comments	**INFO**

This issue collects several items, all superficial, but easy to fix.

- AdminContract:

  ```
  uint256 public constant PERCENT_DIVISOR_DEFAULT = 100;
  // dividing by 100 yields 0.5%
  // Dedaub: No, it yields 1%
  ```

- AdminContract:

  ```
  function setAsDefaultWithRemptionBlock(  // Dedaub: spelling
  ```

- AdminContract:

  ```
  struct CollateralParams {
    …
         uint256 redemptionBlock; // Dedaub: misnamed, it's in seconds
  }
  ```

  (We advise special caution, since the field is set in two ways, so external callers may be confused by the name and pass a block number, whereas the calculation is in terms of seconds.)

- StabilityPool:

```
// Internal function, used to calculcate ...
```

- PriceFeed:

```
* - If price decreased, the percentage deviation is in relation to the the
```

- FeeCollector:

```
function _createFeeRecord(
       address _borrower,
       address _asset,
       uint256 _feeAmount,
       FeeRecord storage _sRecord
    ) internal {
       uint256 from = block.timestamp + MIN_FEE_DAYS * 24 * 60 * 60;
    // Dedaub: `1 days` is the best way to write this, as done
    // elsewhere in the code
```

A19	Opportunities for gas optimization	INFO

Gas savings were not a focus of the audit, but there are some clear instances of repeat work or missed opportunities for immutable fields.

- StabilityPool:

```
function receivedERC20(address _asset, uint256 _amount) external override {
  …
       totalColl.amounts[collateralIndex] += _amount;
       uint256 newAssetBalance = totalColl.amounts[collateralIndex];
  …
}
```

The two highlighted lines (likely) perform two SLOADs and one SSTORE. Using an intermediate temporary variable for the sum will save an SLOAD.

- DebtToken: the following variable is only set in constructor, could be declared immutable.

```
address public timelockAddress;
```

A20	Magic constants	INFO

Our recommendation is for all numeric constants to be given a symbolic name at the top of the contract, instead of being interspersed in the code.

- VesselManagerOperations::getRedemptionHints:

```
collLot = collLot * REDEMPTION_SOFTENING_PARAM / 1000;
```

- AdminContract::setAsDefaultWithRedemptionBlock:

```
if (blockInDays > 14) {  ...
```

- BorrowerOperations::openVessel:

```
contractsCache.vesselManager.setVesselStatus(vars.asset, msg.sender, 1);
// Dedaub: 1 stands for "active", but is obscure
```

A21	I-naming inconsistent	INFO

Contract IDebtToken is not really an “interface”, since it contains full ERC20 implementation functionality.

A22	Maximum allowed deviation between two consecutive oracle prices seems to be too high	INFO

In `PriceFeed.sol` there is a `MAX_PRICE_DEVIATION_FROM_PREVIOUS_ROUND` constant set to 5e17 i.e. 50%. If the percentage deviation of two consecutive Chainlink responses is greater than this constant, the protocol rejects the new price as invalid. But the value of this constant seems to be too high. Moreover, we think it would be better if the protocol used a different `MAX_PRICE_DEVIATION_FROM_PREVIOUS_ROUND` for each collateral asset considering also the volatility of the asset.

A23	Compiler bugs	INFO

The code has the compile pragma ^0.8.10. For deployment, we recommend no floating pragmas, i.e., a fixed version, for predictability. Solc version 0.8.10, specifically, has some known bugs, which we do not believe to 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.