PUBLIC

Code Assessment

of the sBOLD Smart Contracts

May 19th, 2025

Produced for

K3 Capital

by

CHAINSECURITY

Contents

1 Executive Summary

Dear K3 Capital,

Thank you for trusting us to help sBOLD with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of sBOLD according to Scope to support you in forming an opinion on their security risks.

K3 Capital implements sBOLD, an ERC4626 vault that deposits BOLD into Liquity V2 stability pools, providing a tokenization and asset allocation layer on top of Liquity V2 Stability Pool deposits.

The most critical subjects covered in our audit are accounting correctness, reentrancies, and interactions with Stability Pools. In Version 3, security regarding all the aforementioned subjects is high. The possibility of value extraction through self-liquidations has been mitigated. Issues regarding the interaction with Liquity V2 Stability pools that were present in Version 1 have been fully remediated. The logic in Liquity V2 Stability Pools has been modified between Version 2 and Version 3 of this codebase. A minor integration issue is introduced by the new Stability Pool logic (withdraw fail because stability pool cannot be emptied).

The general subjects covered are testing, price conversions, fees, and ERC-4626 compliance. Testing is improvable, a number of issues were uncovered that should have been found through testing. Incorrect price conversions, inconsistente handling of fees, and low ERC-4626 standard compliance have all been addressed and are now appropriate.

In summary, we find Version 3 provides a good level of security.

It is important to note that security audits are time-boxed and cannot uncover all vulnerabilities. They complement but don't replace other vital measures to secure a project.

The following sections will give an overview of the system, our methodology, the issues uncovered, and how they have been addressed. We are happy to receive questions and feedback to improve our service.

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

Below we provide a brief numerical overview of the findings and how they have been addressed.

Critical-Severity Findings	2
• Code Corrected	2
High-Severity Findings	3
• Code Corrected	3
Medium-Severity Findings	8
• Code Corrected	8
Low-Severity Findings	11
• Code Corrected	9
• Acknowledged	2

2 Assessment Overview

In this section, we briefly describe the overall structure and scope of the engagement, including the code commit which is referenced throughout this report.

2.1 Scope

The assessment was performed on the source code files inside the sBOLD repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	December 12th 2024	3506652b868d8196716f27493066eccbffe32830	Initial Version
2	January 12th 2025	adad09d774de31166849d61223c9c4ea73027a9e	Second Version
3	February 7th 2025	e52e34078faa2238846a637cbf6263396e7363e6	third Version

For the solidity smart contracts, the compiler version `0.8.24` was chosen.

The following files are in scope:

- interfaces/IPriceOracle.sol
- interfaces/IRegistry.sol
- interfaces/ICommon.sol
- interfaces/ISBold.sol
- libraries/Common.sol
- libraries/helpers/Decimals.sol
- libraries/helpers/Constants.sol
- libraries/helpers/TransientStorage.sol
- libraries/logic/SwapLogic.sol
- libraries/logic/SpLogic.sol
- libraries/logic/QuoteLogic.sol
- base/BaseSBold.sol
- oracle/pyth/PythOracle.sol
- oracle/Registry.sol
- oracle/chainlink/ChainlinkOracle.sol
- oracle/chainlink/BaseChainlinkOracle.sol
- oracle/chainlink/ChainlinkLstOracle.sol
- sBold.sol

2.1.1 Excluded from scope

Liquity V2 is not in scope. Third party libraries are not in scope and assume to behave correctly according to their specification. Scripts and tests are not in scope.

2.2 System Overview

This system overview describes Version 2 of the contracts as defined in the Assessment Overview.

At the end of this report section, we have added subsections for each of the changes according to the versions.

Furthermore, in the findings section, we have added a version icon to each of the findings to increase the readability of the report.

K3 Capital offers sBOLD, a vault that accrues value on BOLD deposited by users by supplying it to Liquity V2 Stability Pools.

Liquity V2 implements a stablecoin system, where the token is called BOLD, which offers fixed interest rates for borrowers. Low interest borrows can be "redeemed", that is anybody can repay their debt and get a portion of the collateral at 1:1 price (no discount). Borrows that are under-collateralized can be liquidated. The (primary) counterparty of a liquidation is a Stability Pool, a vault containing BOLD deposited by suppliers, and which accumulates collateral from liquidations and BOLD from interest paid by the minters. During a liquidation, BOLD is burned from the stability pool, and collateral from the liquidated borrow is transferred to the Stability Pool, at a discount of 5% on its USD value (liquidations assume BOLD:USD price is 1). The BOLD suppliers in the Stability Pool receive the collateral in proportional amount to their BOLD deposits, and the BOLD burned is taken proportionally from each supplier. BOLD supports three collaterals: WETH, wstETH, and RocketPool ETH. Every collateral has its own Stability Pool. To incentivize liquidity in Stability Pools, 75% of the interest generated by the BOLD stablecoin for each collateral is distributed to the corresponding Stability Pool.

Liquity Stability Pools deposits are not tokenized, Liquidity Providers do not receive a Stability Pool token that is fungible. sBOLD implements a tokenization layer for Stability Pool liquidity: users can deposit BOLD in the sBOLD vault, and receive sBOLD tokens that are fungible and accrue value as the Stability Pool earns interest and is used in profitable liquidations.

For internal accounting, stability pools distinguish between the compounded deposit, the yield, and the pending yield as BOLD balances associated with a position. They also distinguish the stashed collateral and the pending collateral balances.

2.2.1 sBOLD vault

The sBOLD vault deposits BOLD in the three Liquity V2 Stability Pools, in fixed ratios defined at vault initialization. The sBOLD vault implements the ERC4626 tokenized vault interface and emits shares as users deposit BOLD in it (the shares are the sBOLD token).

The following are the main actions performed on the sBOLD vault:

- Deposits: Mint shares through functions `deposit()` and `mint()`.
- Withdrawals: Withdraw assets from shares with `withdraw()` and `redeem()`.
- Swaps: swap collateral for BOLD, realizing a profit/loss for the vault.

2.2.1.1 Deposits

Liquidity providers can deposit BOLD in the sBOLD vault, and receive sBOLD shares in exchange. The deposits are performed with the `deposit()` function, which lets the caller specify an amount of BOLD, and returns the shares minted, or deposits can be performed with the `mint()` function, which allows specifying a desired amount of sBOLD shares and will pull the required BOLD amount. The value of shares is computed from the total amount of BOLD owned by sBOLD. This includes the BOLD balance of the sBOLD contract, BOLD deposited in the Stability Pools, BOLD yield and pending yield of the Stability Pools, and market value of the accumulated collateral in the Stability Pools and sBOLD contract minus a configurable slippage tolerance and fees. If the total value of the collaterals in Stability Pools exceeds the `maxCollInBold` threshold, a value configured by the owner, deposits are suspended.

A fee is levied from the BOLD amount deposited. The fee is set by the vault owner and is stored in the `feeBps` variable. It can be up to 5% of the deposited amount. If the total value of the collaterals exceeds the `maxCollInBold` threshold, deposits are suspended.

The amount of BOLD deposited is supplied to the three Stability Pools, proportionally to the ratios defined at vault initialization. BOLD is supplied to a Stability Pool through function `StabilityPool.provideToSP(amount, false)`. The `false` argument causes the yield accumulated to be deposited as principal in the Stability Pool, and it causes the pending collateral to be stashed.

2.2.1.2 Withdrawals

Shareholders of sBOLD can burn their shares and withdraw the underlying BOLD through functions `withdraw()`, which allows to specify a desired asset amount and burns the corresponding shares, and `redeem()`, in which the amount of shares is specified.

During a withdrawal, the amount of assets withdrawn and corresponding shares burned is first established. The value of the shares is computed from the total amount of BOLD the vault owns, like for the deposit case. If the value of the collaterals owned by sBOLD exceeds `maxCollInBold`, withdrawals are suspended.

From each Stability Pool, a portion of the deposits is withdrawn proportionally to the amount of sBOLD shares burned. Withdrawals from the Stability Pools are performed by calling `StabilityPool.withdrawFromSp(amount, true)`. The `true` flag means that the accumulated yield, and the pending yield, are transferred to sBOLD, and the pending and stashed collateral are also transferred to sBOLD. In general this means that withdrawals from sBOLD will increase the BOLD balance of sBOLD, and decrease the amount of BOLD invested in the Stability Pools.

2.2.1.3 Swaps

Liquidations triggered in Liquity will burn some of the BOLD deposited in a stability pool, and transfer discounted collateral from the liquidated trove to the depositors of the Stability Pool. This means that over time, as liquidations happen, sBOLD will come to own some amount of the three collateral types. The `swap()` function of sBOLD allows to realize the profit (or loss) from the liquidations, by swapping these collateral amounts to BOLD. The vault owner can configure a `swapAdapter`, that is a contract that gets called with data supplied by the caller of `swap()`. During the execution of the `swap()` function, for each collateral the `swapAdapter` is given allowance for the balance of the collateral, it is called with user provided data, and it is checked that the call results in the desired increase in BOLD balance. The value of each collateral is computed by relying on oracles. The swap requires that the proceeds for each collateral are its value in BOLD, as returned by the oracle, minus a relative slippage tolerance defined by the `maxSlippage` variable. `maxSlippage` is a value in BPS set by the owner of sBOLD and can be up to 10%.

From the proceeds of the swaps an additional fee of up to 5% is transferred to a fee receiver configurable by sBOLD owner. A BOLD reward is also transferred to the caller of `swap()`. The reward is defined as a portion of the swap proceeds, stored in the `rewardBps` variable that can be set by the owner of sBOLD to up to 10%.

2.2.1.4 Share pricing

One of the main tasks of sBOLD is to accurately measure how many assets it owns, so that shares can be priced correctly. The `calcFragments()` view method estimates the total assets value by aggregating the value of BOLD held in the vault (`BOLD.balanceOf(this)`, of collaterals held in the vault (`coll.balanceOf(this)`), and BOLD and collaterals held in Stability Pools.

Stability Pools internally keep track, for every user, of:

- deposits (BOLD): the active BOLD amount that gets used to offset debt during liquidation. This amount compounds negatively at each liquidation.
- yield (BOLD): the interest that the Stability Pool receives from BOLD borrowers, and which is distributed pro-rata to Stability Pool deposits.
- pending yield (BOLD): the yield that has accrued in the internal accounting of Liquity, but for which BOLD has not been materially transferred yet. It is realized on every call to the StabilityPool methods `provideToSP()` and `withdrawFromSp()`.
- pending collateral: collateral accumulated since the last collateral claim/stashing
- stashed collateral: whenever `provideToSP()` or `withdrawFromSp()` are called with the `_doClaim` flag false, the pending collateral is zeroed and goes into the `stashedColl` mapping.

These five sources must be aggregated to estimate the holdings of a Stability Pool depositor. Deposits can be queried with the `getCompoundedBoldDeposit()` method of StabilityPool. Yield and pending yield can be queried with `getDepositorYieldGainWithPending()`. Pending collateral is returned by `getDepositorCollGain()`. Stashed collateral is returned by the `stashedColl()` method.

To deposit in a Stability Pool, the `provideToSP()` function is used. To withdraw, the `withdrawFromSp()` function is used. Both functions accept a `_doClaim` boolean argument, which defines how yield and collaterals are treated: If `_doClaim` is true, the yield (realized and pending) and collateral (pending and stashed) are transferred to the depositor. If `_doClaim` is false, yield is deposited in the Stability Pool, increasing the BOLD amount that will offset future liquidations, and pending collateral is stashed.

On withdrawals, if `withdrawFromSp()` is called with a BOLD amount greater than "deposit" size of the depositor, the withdrawal is capped to the current deposit.

Liquidations can be triggered in the Liquity system when a user's Trove falls below the collateralization ratio of 109%. Liquidations will repay a Trove's debt from the StabilityPool BOLD amount, effectively burning the deposits of Stability Pool suppliers, and will deposit discounted collateral from the Trove to the Stability Pool. The collateral value will in general be greater than the debt value, unless a sudden fall in collateral price has caused bad debt to appear. Still, the profit is not realized until the collateral is held in the Stability Pool.

The formula used for the total assets that contribute in the share price is:

$$totalAssets = compoundedDeposits + yield + pendingYield$$

$$+ \sum_{coll} (stashedColl_{BOLD} + pendingCollateral_{BOLD}) * (1 - maxSlippage - swapReward - swapFee)$$

2.2.2 Oracles

sBOLD requires a way to value the collaterals in term of BOLD. This is necessary to set the minimum BOLD proceeds from swapping collateral (slippage tolerance), and to integrate the collateral value in the total assets controlled by sBOLD.

Pricing of the three collaterals is performed through oracles. The contract `Registry` is the entry point for oracle quotes. Registry implements the `IPriceOracle` interface, which exposes the `getQuote(uint256 inAmount, address base)` method and the `isBaseSupported(address base)` method. `isBaseSupported()` returns true if the `base` address

is a token for which an oracle is set in the Registry. Values are quoted in USD, in 18 decimals precision, with the `getQuote()` method. An amount of base token, and the base token addresses are specified in the call.

Registry forwards the `getQuote()` calls to the oracle that's been configured by the Registry owner, through the `setOracles()` function, as the oracle for the `base` token. Three kinds of oracle contracts are defined:

- ChainlinkOracle
- ChainlinkLstOracle
- PythOracle

2.2.2.1 ChainlinkOracle

The `ChainlinkOracle` implements a wrapper around a chainlink feed for a single `base` token. It allows setting a `maxStaleness` parameter which causes the `getQuote()` method to revert if the chainlink feed returns data which have been updated more than `maxStaleness` seconds ago. The chainlink price is multiplied to the `inAmount` argument of `getQuote()`, and the result is rescaled to have 18 decimals.

2.2.2.2 ChainlinkLstOracle

`ChainlinkLstOracle` works similar to `ChainlinkOracle`, but its purpose is to price LST Eth derivatives which have a LST/ETH feed but not a LST/USD chainlink feed. `ChainlinkLstOracle` is configured with two chainlink data feeds, each one with its own `maxStaleness`, one for the ETH/USD price and the other for the LST/ETH price. The answers from the two feeds are combined to derive the LST/USD price. The value of the `inAmount` in USD is then computed by multiplying `inAmount` with the price, and the result is scaled to 18 decimals.

2.2.2.3 PythOracle

Contract `PythOracle` allows reading from a Pyth pull-based price feed. In the constructor, the `pyth` entry point contract and `base` token addresses are set, together with the `maxStaleness` and `maxConfWidth` parameters, and the Pyth `feedId` is specified. Pyth implements pull-based oracles where the price is continuously updated off-chain by a network of verifiers. When a price is needed on-chain, a price update can be permissionlessly triggered by calling `updatePriceFeeds()` on the `pyth` contract. A price feed update is performed by supplying fresh (more recent than the latest update) signed data to Pyth and paying an update fee. The price update to the Pyth onchain contract happens outside of the sBOLD system. The `PythOracle` contract however checks that the last price reported by the Pyth oracle is more recent than the `maxStaleness`, which is configurable by the owner to up to 15 minutes. Pyth price feeds return a structure composed of `price`, `conf` (confidence interval), `expo` (exponent) and `publishTime`. The price precision is given by `expo`, such that the actual price is `price * 10**expo`. `conf` represents the uncertainty around the price, and has the same precision as `price`. `PythOracle` checks that the `conf/price` ratio does not exceed the `maxConfWidth` limit, which is in BPS.

2.2.3 Privileged Roles

Contracts `sBOLD` and `Registry` are `Ownable`.

The `owner` of `sBOLD` can:

- `setPriceOracle()`: sets the `priceOracle` variable, which is the entry point for oracle calls (expected to be the `Registry` contract)
- `setVault()`: sets the receiver of deposit fees and swap fees.
- `setFees()`: sets the deposit fee (`feeBps`) and the swap fee (`swapFeeBps`).
- `setReward()`: sets the reward share of the swap proceeds.
- `setMaxSlippage()`: sets how much slippage is tolerated by swaps.

- `setSwapAdapter()`: sets the contract which is receives collateral allowance and is called to perform the swap.
- `setMaxCollInBold()`: sets the `maxCollInBold` variable, which defines the threshold of collateral value above which deposits and withdrawals are blocked.
- `pause()` and `unpause()`: the withdrawal and depositing functionality, and the swap functionality can be pause and unpaused.
- `rebalanceSPs()` to change the list of stability pools that the vault deposits into, and their weights.

The `owner` of `Registry` can call `setOracles()`, which sets the oracle contract to which `getQuote(amount, base)` calls are forwarded for a given `base`, by writing it in the `baseToOracle` mapping.

The `owner` of `sBOLD` or `Registry` can also transfer the ownership of the contracts.

2.2.4 Differences with Version 2

In Version 2 the collateral value below `maxCollInBold` is not counted towards the total assets, as it is considered unrealized gains. In Version 3, the whole collateral amount is considered as for the valuation of sBOLD.

2.2.5 Differences with Version 1

In Version 1, `swap()` does not allow partial swaps or swapping only a subset of collateral types. Reentrancy is not blocked on state-modifying functions. `rebalanceSPs()` is not available, so the stability pools and their weights are fixed at deployment.

2.2.6 Trust model and Caveats

The `owner` of `sBOLD` and `Registry` is fully trusted, and assumed to only set parameters that allow the correct functioning of the system and do not damage users.

The fee vault owner is trusted not to steal the fee. If they are in a privileged position to drain the protocol it is considered a bug, but with lower likelihood than if it was an untrusted user.

It is assumed that oracle operators report accurate and timely prices.

The system is assumed to be deployed against the canonical deployment of Liquity V2. The collateral tokens are `wstETH`, `rETH`, and `WETH`

3 Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable the discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts defined in the engagement letter. We assessed whether the project follows the provided specifications. These assessments are based on the provided threat model and trust assumptions. We draw attention to the fact that due to inherent limitations in any software development process and software product, an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third-party infrastructure which adds further inherent risks as we rely on the correct execution of the included third-party technology stack itself. Report readers should also take into account that over the life cycle of any software, changes to the product itself or to the environment in which it is operated can have an impact leading to operational behaviors other than those initially determined in the business specification.

4 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- *Likelihood* represents the likelihood of a finding to be triggered or exploited in practice
- *Impact* specifies the technical and business-related consequences of a finding
- *Severity* is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

Likelihood	Impact		
	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

5 Open Findings

In this section, we describe any open findings. Findings that have been resolved have been moved to the Resolved Findings section. The findings are split into these different categories:

- Security: Related to vulnerabilities that could be exploited by malicious actors
- Design: Architectural shortcomings and design inefficiencies
- Correctness: Mismatches between specification and implementation

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	2

5.1 Withdraw Fail Because Stability Pool Cannot Be Emptied

Design Low Version 3 Acknowledged

CS-SBOLD-029

In March 2025 an upgraded version of the Liquity V2 stability pool has been published, which contains an additional check that the total bold deposits in the Stability Pool do not go below 1 BOLD (see code on GitHub). Withdraws from sBOLD that would leave a stability pool with less than 1 BOLD will revert because of this condition, which is not checked by the `maxRedeem()` and `maxWithdraw()` functions.

The client chose not to address this issue by changing the code. It is unlikely it will ever materialize in practice and in case it causes reverts, it can be remediated by depositing 1 BOLD in the stability pool.

5.2 Profit Can Be Extracted Through Slippage Tolerance

Design Low Version 1 Acknowledged

CS-SBOLD-030

Swapping is by default unpermissioned, and the caller of `swap()` is able to extract part of the value of the collateral because of the slippage tolerance (`maxSlippage`). For example, if `maxSlippage` is configured as 1%, the caller of `swap()` can keep 1% of the value of the collateral for himself.

An attacker could exploit this fact to repeatedly extract value from the protocol, by converting the BOLD in the Stability Pools into collateral, by generating liquidations, and then converting the collateral back to

BOLD extracting the slippage. This could be perfromed repeatedly, to drain the protocol. The profitability of the attack depends on the share of the stability pool controlled by sBOLD, and the `maxSlippage` parameter.

Let's denote `maxSlippage` as m, and the liquidation bonus of BOLD as L. An attacker generating a liquidation on a debt of size B incurs a loss of BL, and the stability pools receives $B(1+L)$ value of collateral. If sBOLD owns the whole stability pool, this means that the amount of profit that can be extract by the attacker in terms of slippage by calling `swap()` is equal to $mB(1+L)$ (m is max slippage).

Therefore, the attack is unprofitable if

$$mB(1+L) \leq BL$$

simplifying:

$$m \leq \frac{L}{1+L}$$

Therefore, the max slippage parameter must be set lower than $\frac{L}{1+L}$ to ensure that the attack strategy is unprofitable. The liquidation penalty L is expected to be set to 5% in Liquity V2.

In the updated version of Liquity BOLD, generating self-liquidations has been rendered less practical. The issue is therefore mitigated.

6 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the Open Findings section.

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	2

- Reentrancies From Swap() Code Corrected
- swap() Calls Incorrect Function in SP to Receive Collateral Code Corrected

High-Severity Findings	3

- Collateral Value Is Not Accounted Below maxCollInBold Threshold Code Corrected
- Incorrect Quote Calculation Code Corrected
- Stashed Collateral Not Taken Into Account Code Corrected

Medium-Severity Findings	8

- provideToSP() Can Be Called With Zero Amount Code Corrected
- BOLD Yield Is Never Redeposited Code Corrected
- Early Return in withdrawFromSP Code Corrected
- Inconsistent Fee Calculation Code Corrected
- Partial Swaps Are Not Allowed Code Corrected
- Undiscounted Collateral Used in _maxCollInBold Code Corrected
- amountProRata Withdrawn From SP Is Too High Code Corrected
- totalAssets() Is Not Overridden Code Corrected

Low-Severity Findings	9

- Incorrect Transient Storage Slot Code Corrected
- Dead Shares Can Be Invested Code Corrected
- ERC4626 Function maxDeposit Should Reflect Pausability Code Corrected
- Inconsistent Decimal Adjustments Code Corrected
- Incorrect Rounding Code Corrected
- Rounding in withdrawFromSP Code Corrected
- Unable to Swap a Single Collateral Code Corrected
- maxDeposit() Should Never Revert Code Corrected
- swap() Reverts if rewardBps Is Set to 100% Code Corrected

Informational Findings	2

- Gas Optimizations Code Corrected
- Unstated Owner Privileges Code Corrected

6.1 Reentrancies From Swap()

Security | Critical | Version 1 | Code Corrected

CS-SBOLD-001

The `swap()` function does not guard against potential reentrancy. Since the `swapAdapter` is called with user provided data, it must be assumed that the caller will be able to execute untrusted code within the context of the external call to `swapAdaper`. In general, calls to swap adapters always have the potential to give control flow to the attacker. Even a legitimate swap adapter may offer callback hooks, or be forced to route through a malicious token. For example, the 1inch router calls a user-provided contract in its `swap()` function. In general, swap routers will allow the call to execute arbitrary code in the form of a malicious token in the swap path: If token A has to be exchanged for token B, the caller can deploy pools for his own token E, such that the swap path will be A -> E -> B. Swapping to E will allow the caller to execute arbitrary code because token E is interacted with.

Being able to execute arbitrary code within the call to `swapRouter` allows two reentrant paths in `sBOLD` that are security critical:

1. The balance check in `SwapLogic._execute()`, which checks the difference in BOLD balance before and after the call to `swapAdapter`, can be tricked by reentering to a function that increases the BOLD balance of the `sBOLD` contract. The `withdraw()` and `redeem()` functions do that, they increase the BOLD amount sitting in the `sBOLD` contract, because the call `SP.withdrawFromSP()` with the `_doClaim` flag set to `true` will transfer all the yield to `sBOLD`. Not all of this collected BOLD is transferred out to the caller of `withdraw()` or `redeem()`, so when the function returns the overall BOLD balance in `sBOLD` has increased. This increase can be stolen by an attacker, since the balance check in `_execute()` will attribute the increase to the swap of collateral to BOLD, even if it actually comes from balance owned by `sBOLD`.
2. Reentering the `calcFragments()` and `getSBoldRate()` functions within a swap can return invalid results, as an inconsistent state is read: collateral has already been sent out of the contract, but BOLD has not have been received yet. This causes `calcFragments()` and `getSBoldRate()` to underestimate the total assets / share price. As a result, it is possible to `deposit()` at a discount with a reentrant call.
3. Additionally, if the `sBOLD` token is being used in external systems, for example as collateral, this conversion rate manipulation could allow an attacker to trigger malicious liquidations on loans that are actually healthy. This is a read-only reentrancy: only view functions are called on sBOLD itself while the system is in an inconsistent state.

Code corrected:

In Version 2, reentrancy guards have been applied on unpermissioned state modifying functions (`deposit`, `mint`, `redeem`, `withdraw`, `swap`, `rebalanceSPs`). In Version 3, read-only reentrancy guards have been added to view functions that expose the internal state of `sBOLD`.

6.2 `swap()` Calls Incorrect Function in SP to Receive Collateral

Correctness | Critical | Version 1 | Code Corrected

CS-SBOLD-002

`swap()` calls `SP.claimAllCollGains()`, which in Liquity reverts if the depositor has a non-zero BOLD balance, see https://github.com/liquity/bold/blob/69d715c986789fe35aca7af06b5671d6e25972d0/contracts/src/StabilityPool.sol#L354.

As a result, `swap()` will be unavailable every time it is called. To receive collateral from the the StabilityPool, `provideToSP()` or `withdrawFromSP()` can be used with the `_doClaim` flag set to true.

Independently fixed by K3 Capital:

K3 Capital independently found the issue while the audit was ongoing, before it was also independently reported by ChainSecurity. The invalid call to `SP.claimAllCollGains()` was replaced with `SP.withdrawFromSP(0, true)`. The new call performs a 0-size withdrawal with bool `_doClaim` flag equal to `true`, which transfers all the BOLD yield and collateral earned in the StabilityPool to the `sBOLD` contract.

6.3 Collateral Value Is Not Accounted Below `maxCollInBold` Threshold

Design | High | Version 1 | Code Corrected

CS-SBOLD-008

The first return value of function `calcFragments()` is the amount of assets used for share pricing. It includes the amounts of BOLD deposited and earned in the Stability Pools, but it does not account for the value of the collateral up to `maxCollInBold`. This is the intended design, however it implies value transfer from existing users to new users, who can mint sBOLD at a discount and redeem it at the full value.

Assuming `maxCollInBold` is 7.5K, this means that collateral owned by sBOLD will not be accounted towards sBOLD total assets up to a value of 7.5K BOLD. Starting from an sBOLD state with 0 collateral value, if a liquidation happens that converts 7.5K BOLD into collateral (with the same value), the total assets will now 7'500 BOLD lower. The token value for existing depositors has gone down. The collateral value of 7.5K can be extracted by minting a large amount of sBOLD shares, obtained at a discount, executing a successful `swap()`, which exchanges collateral to BOLD such that it is accounted again, and then redeeming the shares at the full value.

A quirk in BOLD behavior allows users to create BOLD troves that can be immediately liquidated: see issue `CS-BOLD-019` in ChainSecurity BOLD report . Malicious users can exploit this quirk in the core BOLD protocol to repeatedly self-liquidate at a loss and extract value from sBOLD to cover the loss, potentially draining sBOLD. The exploit steps are the following:

1. Attacker `A` opens a trove with collateral `C`. `A` deposits collateral and obtains BOLD.
2. Attacker `A` triggers a liquidation on their trove: `A` incurs a liquidation loss of 5%. The Stability pool burns BOLD and absorbs collateral from `A`.
3. The total assets accounted by sBOLD drop by the amount of collateral received from the liquidation.
4. `A` mints a large amount of sBOLD shares at a discount.
5. `A` execute the `swap()` function of sBOLD, which brings the total assets back to the full value and increases the share price.
6. `A` withdraws the shares, extracting much of the value increase from the swap at step 5.
7. `A` repeats the process.

The attack can be mathematically modeled as follows:

Consider the variables `t`, that is the `maxCollInBold` sBOLD collateral threshold (7.5K), `w`, the share of stability pool liquidity owned by sBOLD for the target collateral, and `p`, the liquidation penalty applied by BOLD (5%). The attacker's debt in the trove is `L`.

Then, the attacker's (self-)liquidation loss is Lp. The change in collateral in sBOLD is $\Delta_{coll} = L(1+p)w$, and the change in BOLD in sBOLD is $-Lw$

Assuming sBOLD has no collateral balance in the beginning, the change in collateral has to be lower than the threshold t such that `deposit()` operations are still possible after the change in collateral in sBOLD:

$$\Delta_{coll} \leq t \Leftrightarrow L \leq \frac{t}{(1+p)w}$$

This relation tells us the most debt L that the attacker can self-liquidate without blocking deposits and withdrawals in sBOLD because of failing collateral health checks. This limits the size of each iteration of the attack, however the attacker can simply perform more iterations.

For the attack to be profitable, the liquidation loss of the attacker Lp must be less than the value they extract from sBOLD at steps 4-6. Assuming the attacker can extract all the collateral, this value is the Δ_{coll} :

$$Lp \leq L(1+p)w$$

Simplifying and rearranging the terms, this shows the minimal conditions for the attack to be profitable:

$$w \geq \frac{p}{1+p}$$

Given the liquidation penalty applied by BOLD of 5%, this means that the attack is profitable if sBOLD controls more than 3.3% of the Stability Pool's liquidity for the given collateral.

In practice, not all value can be extracted at steps 4-6 without access to unlimited capital. But if the attacker can supply to sBOLD a large amount of BOLD, obtained for example with a flashloan, such that they can extract a ratio α of the value increase, the profitability inequality (loss less than profit) can be adjusted as follow:

$$Lp \leq \alpha\Delta_{coll} \Leftrightarrow w \geq \frac{p}{(1+p)\alpha}$$

If for example α is 50%, the attack is profitable as soon as sBOLD provides 6.6% of the total Stability Pool liquidity for the given collateral.

Further simplification has been made by ignoring the deposit fee at step 4, ignoring the cost of capital, and ignoring the gas fees in the attacker's losses. These factors will concur to reduce the parameter space where the attack is profitable. However they do not address by themselves the root cause of the issue, which is ignoring the collateral value up to `maxCollInBold`.

Code corrected:

In Version 3, the collateral value is included in the valuation of the sBOLD vault.

6.4 Incorrect Quote Calculation

Correctness | High | Version 1 | Code Corrected

CS-SBOLD-005

`QuoteLogic.getInBoldQuote()` and `sBold._calcCollValue()` multiply the price of one BOLD in USD with the collateral price in USD.

```
boldUnitQuote.mulDiv(collQuote, 10 ** Constants.ORACLE_PRICE_PRECISION)
```

This is incorrect and does not result in a quote for the price of the collateral in BOLD. For example, if BOLD has depegged and is now worth $0.5, the collateral quoted in USD should be multiplied by 2, not multiplied by 0.5, to get the BOLD quote.

This can result in inaccurate valuation of sBold assets, and inaccurate slippage protections being applied.

Code corrected:

In Version 2, the calculation has been fixed in both spots such that `boldUnitQuote` is the denominator.

6.5 Stashed Collateral Not Taken Into Account

Design High Version 1 Code Corrected

CS-SBOLD-006

In `StabilityPool`, if a depositor alters their position while setting `_doClaim` to false, the stability pool, for accounting purposes, moves the collateral gains internally accumulated to the *stashed* collateral variable. `SpLogic._getCollBalanceSP()` does not take stashed collateral into account, as it only queries the collateral gain through `sp.getDepositorCollGain(addr)`, which does not include stashed collateral, which is returned by `sp.stashedColl(addr)`.

In particular, `SpLogic.provideToSP()`, which is called by `deposit()` and `mint()`, sets `_doClaim` to false, and as a result can stash collateral.

Since stashed collateral is not taken into account, the true collateral balance of sBOLD might be considerably under-estimated: shares could be acquired at a significant discount, and then redeemed at the actual value after a `swap()`.

Code corrected:

In Version 2, stashed collateral is correctly accounted for.

6.6 `provideToSP()` Can Be Called With Zero Amount

Correctness Medium Version 2 Code Corrected

CS-SBOLD-022

The `provideToSP()` function in stability pools, called at line 24 of `SpLogic.sol`, will revert if its amount argument is zero.

When withdrawing, sBOLD will attempt to re-deposit any accumulated yield to the stability pools. If two withdrawals occur in the same block, the first one will claim all the yield, and no yield will accrue in time for the second deposit. Thus, the argument will be zero and the second withdrawal will be blocked.

`0` value calls to `SP.provideToSP()` can also happen when a small `assets` amount rounds to zero at line 22 of `SpLogic.sol`.

Code corrected:

In Version 3, the call to `SP.provideToSP()` is omitted if the argument is zero.

6.7 BOLD Yield Is Never Redeposited

Design | Medium | Version 1 | Code Corrected

CS-SBOLD-007

When withdrawing from a Stability Pool in `SpLogic.withdrawFromSP()`, the yield accumulated in the Stability Pools is pulled into the contract's internal balance, since `StabilityPool.withdrawFromSP()` is used with `_doClaim == true`. This result in BOLD sitting idle in the balance of sBOLD. It then does not accrue interest and there is no mechanism to re-deposit it. Over the lifetime of the sBold contract, more and more BOLD will end up idle in the balance of sBold instead of being invested in the Stability Pools.

Code corrected:

In Version 3, the `provideToSP()` call redeposits the yield to the SPs.

6.8 Early Return in withdrawFromSP

Correctness | Medium | Version 1 | Code Corrected

CS-SBOLD-003

The loop in `SpLogic.withdrawFromSP()` contains a return statement that is executed if one of the pro-rata amounts is equal to zero. This causes the function to skip any later stability pools, even though those may have non-zero pro-rata amounts to contribute to the withdrawal.

Code corrected:

In Version 3, a `continue` statement is used if both the deposits and yield in a given stability pool are zero, so that the function can attempt to withdraw from the remaining stability pools.

6.9 Inconsistent Fee Calculation

Correctness | Medium | Version 1 | Code Corrected

CS-SBOLD-004

Fees are computed inconsistently between `deposit()` and `mint()`.

Let $0 < f < 0.05$ be the fee rate.

When calling `deposit()`, the user pays A worth of assets to receive $(1-f)A$ worth of shares. When calling `mint()`, the user pays A worth of assets to receive $\frac{1}{1+f}A$ worth of shares.

This discrepancy is because in `deposit()` case, the fee is removed from the input by multiplying with $1-f$, whereas in the `mint()` it is added to the output by multiplying by $1+f$. Since $\frac{1}{1+f} > 1-f$ for $f>0$, it is always more advantageous for the user to use `mint()` instead of `deposit()`, as they will pay less fees.

Another consequence is that in `mint()`, even though the fee paid by the user less than in `deposit()`, the same amount is sent to the fee receiver. The actual fee paid is $\frac{f}{1+f}A$ but `sBOLD` sends fA to the fee receiver. The difference f^2A is paid by all shareholders. This leads to a potential draining attack from the fee vault administrator: The administrator first mints a great amount of shares, paying the fee to themselves and receiv ing the excess fee out of the vault depositors' assets. They can then instantly withdraw from the vault and recover their assets with a profit.

Code corrected:

In Version 2, the fee is now transfered directly from the depositor to the fee vault. As a result, the vault no longer deducts the fee from the assets it receives. Thus the fee vault administrator no longer represents a threat. However, a discrepancy between `deposit()` and `mint()` is still present.

In Version 3, the fee calculation for ERC-4626 was corrected to use the strategy suggested in https://docs.openzeppelin.com/contracts/4.x/erc4626#fees. Functions `deposit()` and `mint()` therefore apply the same fee consistently.

6.10 Partial Swaps Are Not Allowed

Design Medium Version 1 Code Corrected

CS-SBOLD-009

When calling `swap()`, the `swapAdapter` is expected to swap the entire outstanding balance of the collateral. It is not possible to perform a partial swap. In a situation where the size of the collateral amount makes swaps exceed the slippage tolerance limit, no collateral can be swapped. At the same time, redemption and deposits will be automatically paused due to the protocol mechanic (`maxCollInBold` threshold) meaning that funds are stuck.

Code corrected:

Partial swaps are supported in Version 2.

6.11 Undiscounted Collateral Used in `_maxCollInBold`

Correctness Medium Version 1 Code Corrected

CS-SBOLD-011

`calcFragments()` subtracts the raw valuation of the recovered collateral from the sBOLD value:

```
uint256 _maxCollInBold = maxCollInBold > collInBold ? collInBold : maxCollInBold;

return (totalBold - _maxCollInBold, boldAmount, collValue, collInBold);
```

`totalBold` is defined as `boldAmount + collInBoldNet`, which includes the pessimistic net expected value of the collateral, (after swap fee, reward, and max slippage). However `_maxCollInBold` which is subtracted from it uses the `collInBold` value, which is higher than `collInBoldNet`.

This leads to the value in the protocol being underestimated in realistic circumstances.

In some circumstances, if `boldAmount` is lower than the difference between `collInBoldNet` and `collInBold`, the subtraction `totalBold - _maxCollInBold` can revert, breaking the deposit and withdraw functionality.

Code corrected:

In Version 3, the collateral value is no longer removed from the total BOLD value, so the issue is fixed.

6.12 `amountProRata` Withdrawn From SP Is Too High

Design | Medium | Version 1 | Code Corrected

CS-SBOLD-012

`withdrawFromSP()` in `SpLogic` choses the withdrawal amount based on the total BOLD amount of the StabilityPool (`_getBoldAssetsSP()`), and the shares that are being withdrawn by the caller. This causes `SP.withdrawFromSP(amountProRata, true)` to be called, with the withdrawal amount set to `amountProRata`. However, the withdrawal amount passed to `SP.withdrawFromSP()` is the amount that gets withdrawn from the StabilityPool deposits only, not including yield, but `amountProRata` is computed including yield. Since `SP.withdrawFromSP()` is called with `_doClaim` set to `true`, the whole yield is transferred because of this (not just the withdrawer's yield). This overall results in more BOLD being withdrawn than needed for the shares redemption, resulting in idle BOLD balance in `sBold`.

Code corrected:

In Version 2, `withdrawFromSP()` processes the yield after withdrawing it, keeping a pro-rata share of it for the caller and redepositing the remainder.

6.13 totalAssets() Is Not Overridden

Correctness | Medium | Version 1 | Code Corrected

CS-SBOLD-013

`sBold` derives from OpenZeppelin `ERC4626`. The `totalAssets()` function is not overridden in the `sBOLD` contract. As a result the the default implementation is used, defined in the OpenZeppelin library, which returns the BOLD balance of the contract.

The BOLD balance of the contract does not represent `totalAssets()`, as it is usually much smaller, and does not correspond to the ERC-4626 specification.

Code corrected:

In Version 2, `totalAssets()` is overridden.

6.14 Incorrect Transient Storage Slot

Correctness | Low | Version 2 | Code Corrected

CS-SBOLD-025

In `TransientStorage.loadCollValue()`, the constant `COLLATERAL_IN_BOLD_STORAGE` is used for the transient storage slot instead of `COLLATERAL_VALUE_STORAGE`. Since the return value is not used internally in the current state of the codebase, this only impacts the `calcFragments()` public view function.

The constant has been corrected in Version 3.

6.15 Dead Shares Can Be Invested

Correctness | Low | Version 1 | Code Corrected

CS-SBOLD-024

In `swap()` and `rebalanceSPs()`, BOLD sitting in the sBold contract is resupplied to the stability pools.

However, 1 BOLD (`10**18`) which belongs to the dead shares is by convention kept in the contract.

The amount to provide is calculated as:

```
assetsToProvide = assetsInternal > deadShareAmount ? assetsInternal - deadShareAmount : assetsInternal
```

in both `swap()` and `rebalanceSPs()`. If the balance of the contract is actually less than 1 BOLD, the whole balance (`assetsInternal`) will be redeposited to the SPs, instead of keeping it in the contract.

Code corrected:

In Version 3, both functions ensure 1 BOLD is kept uninvested in the contract.

6.16 ERC4626 Function maxDeposit Should Reflect Pausability

Design | Low | Version 1 | Code Corrected

CS-SBOLD-014

Functions `maxDeposit()`, `maxMint()`, `maxWithdraw()`, and `maxRedeem()` should take into account the `paused` state of the system. That is, they should return 0 if the system is paused.

Code corrected:

In Version 2, all the aforementioned functions correctly return 0 when the system is paused.

6.17 Inconsistent Decimal Adjustments

Correctness | Low | Version 1 | Code Corrected

CS-SBOLD-015

Decimals adjustments are used inconsistently, but without consequences with the choice of tokens in use for the deployment, in the following cases:

1. Line 140 sBold.sol: `decimals()` returns the decimals of the shares, not of the underlying asset. They happen to be the same since `_decimalsOffset()` is set to 0.
2. `Decimals.scale()` is used in sBold.sol at line 257, and at line 48 of SwapLogic.sol, without any effect.
3. in BaseChainlinkOracle.sol line 48: `_decimals` is used instead of `ORACLE_PRICE_PRECISION`
4. ChainlinkLstOracle.sol line 51: divides by `ORACLE_PRICE_PRECISION` instead of `base.decimals()`.
5. PythOracle.sol line 101: `scale` should be `ORACLE_PRICE_PRECISION`, and should divide by `baseDecimals` which is the decimals of `inAmount`.

Code corrected:

All the points of this issue have been addressed.

6.18 Incorrect Rounding

Correctness | Low | Version 1 | Code Corrected

CS-SBOLD-016

The `_convertToAssets()` function uses `getSBoldRate()`, which always rounds down, to compute the conversion rate. As a result, called in `previewMint()` with `Rounding.Ceil`, it may actually round down.

The `_convertToShares()` function uses `getSBoldRate()` in its denominator, which can lead to the opposite problem: `previewDeposit()` and `convertToShares()` could exhibit the wrong behavior.

ERC-4626 vaults must always round in favor of the protocol. In this particular case, this leads to an attack which is unlikely to be profitable: Say the deposit fee is zero and the user calls `mint()` asking for 500 shares and the `getSBoldRate()` is off-by-one. In `_convertToAssets()`, the expression `shares.mulDiv(getSBoldRate(), 10 ** decimals(), rounding)` could amplify the rounding error up to `shares / 10 ** decimals()` in favor of the user. The profits from this attack is unlikely to outweigh gas costs.

Code corrected:

In Version 2, a new function has been introdcued: `_getSBoldRateWithRounding(rounding)` which returns the conversion rate with the desired rounding. In `_convertToShares()` it is used incorrectly. The rounding of the rate is in the same direction as the overall rounding, but the rate is at the denominator, so the rounding of the rate should be in the opposite direction.

In Version 3, the rounding mode is inverted in the denominator, therefore correcting the remaining issue.

6.19 Rounding in withdrawFromSP

Correctness | Low | Version 1 | Code Corrected

CS-SBOLD-026

`SpLogic.withdrawFromSP()` does not take rounding errors into account.The computation of `portion` can round down.

Say there are 100 shares and 300 assets. A withdrawal for 100 assets it requested. Assuming ERC-4626 conformance, `previewWithdraw()` returns 33 shares to `withdraw()` which passes it to `SpLogic.withdrawFromSP()`. Then, `portion` will be `33 * 10**16`. Assuming for simplicity that there is a single stability pool, then `amountProRata = 300 * 33 * 10**16 / 10**18 = 99`. Thus the withdraw will be 1 wei short.

Under normal circumstances, the sBold contract should have some balance of BOLD that can cover the extra wei, both due to the dead share and to other rounding events.

Code corrected:

The logic of `withdrawFromSP()` has been significantly changed in Version 3 and the issue is not present anymore.

6.20 Unable to Swap a Single Collateral

Design Low Version 1 Code Corrected

CS-SBOLD-017

When calling `swap()`, users are forced to provide swap instructions for all 3 collateral types. It is not possible to skip a collateral, unless its balance is 0. If even one oracle is unavailable swapping is blocked for all 3 collateral. If it reports too high a price, swapping is unprofitable. If dust amounts of one collateral are present in the contract, they will have to be swapped anyway. This could result in failure to swap, as certain AMMs might revert for swap amounts that are too low. For example, it is not possible to swap 1 wei of WETH for USDC because the minimum amount of USDC received (1 wei) is worth more than 1 wei of WETH, since USDC has fewer decimals. So if USDC is in the swap path between WETH and BOLD the swap could revert.

Failure to swap can potentially lead to untimely swaps and losses, and to deposits and withdrawals being inaccessible while the system is locked if the `maxCollInBold` threshold is exceeded.

Code corrected:

In Version 2, `swap()` callers can choose which collateral types to swap.

6.21 `maxDeposit()` Should Never Revert

Design Low Version 1 Code Corrected

CS-SBOLD-018

Functions `maxDeposit()`, `maxMint()`, `maxWithdraw()`, and `maxRedeem()` should never revert and return 0 instead. In the current system, they could revert because of stale or invalid oracle results.

Code corrected:

In Version 2, all the aforementioned functions return 0 when no oracle quote is available.

6.22 `swap()` Reverts if `rewardBps` Is Set to 100%

Design Low Version 1 Code Corrected

CS-SBOLD-019

Function `setReward()` allows setting the portion of the swap that goes to the caller to up to 100%. However if 100% of the swap proceeds go to the caller, no amount is left to be deposited to the Stability Pools. The `SpLogic.provideToSP()` function will be called with `0` as the asset parameter, causing a revert of the `swap()` because calling `StabilityPool.provideToSP()` with a 0 amount is not allowed.

Code corrected:

In Version 2, the `rewardBps` parameter is capped to 10%.

6.23 Gas Optimizations

Informational Version 1 Code Corrected

CS-SBOLD-021

There are several spots where gas consumption can be improved. Following is a non-exhaustive list of them:

1. In `withdraw()` and `redeem()`, `_checkCollHealth()` is called once directly and once through `maxWithdraw()` or `maxRedeem()`. One of the calls is essentially redundant, although the revert reason may change if it is removed.
2. In `swap()`, the `sps` array is copied from storage into memory three times: for the call to `claimAllCollGains()`, for the call to `getCollBalances()`, and for the call to `provideToSP()`. It would be cheaper to reuse a single copy.
3. The `Registry.setOracles()` external function could keep `oracles` in calldata to avoid copying it to memory.
4. The `weight` field in the `ISBold.SP` structure could be of type `uint96` or smaller to benefit from storage packing.
5. The `Feed` struct defined in `BaseChainlinkOracle` could optimized to use 1 storage slot instead of 2. Like-wise, struct SP defined in `ISBold` uses 3 storage slots instead of 2.

Code corrected:

Number 2 not fully implemeneted.

6.24 No Rebalancing Between Stability Pools

Note Version 1 Code Corrected

CS-SBOLD-027

The `weights` configured in `sBold` constructor are expected to define the capital allocation between the different pools, but they are only used when depositing BOLD in the stability pools during `deposit()` and `mint()`, and after `swap()`. There is no guarantee that these ratios are maintained: if liquidations target a stability pool more than the others, its BOLD deposits will decrease below the `weights` allocation. Like-wise, if a pool earns more interest than the others, it will grow bigger than its configured weight allocation.

Over the lifetime of `sBold` this can lead to a significant discrepancy from the configured weights with no possibility of rebalancing.

Code corrected:

In Version 2, the owner can pass the initial configuration to `rebalanceSPs()` to rebalance.

6.25 Unstated Owner Privileges

Informational Version 1 Code Corrected

CS-SBOLD-023

1. The owner can drain `sBold` completely by setting `swapAdapter == StabilityPool`, so that `_execute()` can call `withdrawFromSP()` to withdraw all the BOLD from the stability pool, and then set `swapAdapter == bold` such that `_execute()` can call `BOLD.approve(owner, uint256.max)`. The owner can perform all the required steps in a single transactions and completely empty the contract.
2. The owner can change the price oracles, such that the collateral pricing can be manipulated arbitrarily.
3. The owner can block deposits and swaps by setting the fee receiver address to the contract's address, causing fee payments to revert since the BOLD token contract will prevent transfers of BOLD to itself.
4. The owner can block swaps: set the swapper reward to 100% and the swap fee to a non-zero value.
5. The owner can block deposits, but allow swaps and withdrawals: set `vault = bold` and `swapFeeBps = 0`, such that deposits fail because the deposit fee can't be transferred to the BOLD address, but swap succeed because the fee is disabled there.

Code corrected:

The highlighted issues have been addressed. The owner however remains a privileged role that can is trusted to behave correctly not to damage the users of the system.

6.26 Weights Cannot Be Modified Once Set

Note Version 1 Code Corrected

CS-SBOLD-028

Fixed weights for capital allocations are defined in `sBold`, which set the proportions with which the BOLD deposited in `deposit()` and `mint()`, or obtained through `swap()`, is deposited in the several stability pools. While the `weights` defined at deployment time might be suboptimal in terms of capital allocation, `sBold` does not provide a way to change these proportions. In particular, Liquity V2 defines a procedure to shutdown a collateral branch in certain circumstances. The disabled collateral branch will earn no interest, but `sBold` will keep allocating BOLD to its Stability Pool.

Code corrected:

In Version 2, the `rebalanceSPs()` function allows the owner to change the allocation.

7 Informational

We utilize this section to point out informational findings that are less severe than issues. These informational issues allow us to point out more theoretical findings. Their explanation hopefully improves the overall understanding of the project's security. Furthermore, we point out findings which are unrelated to security.

7.1 Collateral Counted Twice if Collaterals Repeated

Informational Version 1 Acknowledged

CS-SBOLD-020

If BOLD is deployed with repeated collateral (WETH collateral present for 2 different branches, e.g. one with 110% LTV and one with 150% LTV), the collateral balance calculation double counts the balance of the collateral held in the contract. This should not happen for the official deployment of BOLD on Ethereum mainnet, which has WETH, wsETH, and rETH as the only 3 branches, but could prevent the `sBOLD` codebase from being used with forks of Liquity that use a different configuration.