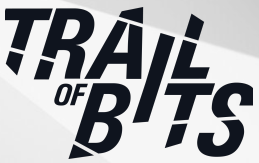

Salty.IO

Security Assessment

December 13, 2023

Prepared for:
Daniel Cota
Salty.IO

Prepared by: **Richie Humphrey and Damilola Edwards**

About Trail of Bits

Founded in 2012 and headquartered in New York, Trail of Bits provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code. With 100+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

We maintain an exhaustive list of publications at https://github.com/trailofbits/publications, with links to papers, presentations, public audit reports, and podcast appearances.

In recent years, Trail of Bits consultants have showcased cutting-edge research through presentations at CanSecWest, HCSS, Devcon, Empire Hacking, GrrCon, LangSec, NorthSec, the O'Reilly Security Conference, PyCon, REcon, Security BSides, and SummerCon.

We specialize in software testing and code review projects, supporting client organizations in the technology, defense, and finance industries, as well as government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, and Zoom.

Trail of Bits also operates a center of excellence with regard to blockchain security. Notable projects include audits of Algorand, Bitcoin SV, Chainlink, Compound, Ethereum 2.0, MakerDAO, Matic, Uniswap, Web3, and Zcash.

To keep up to date with our latest news and announcements, please follow @trailofbits on Twitter and explore our public repositories at https://github.com/trailofbits. To engage us directly, visit our "Contact" page at https://www.trailofbits.com/contact, or email us at info@trailofbits.com.

Trail of Bits, Inc.
228 Park Ave S #80688
New York, NY 10003
https://www.trailofbits.com
info@trailofbits.com

Notices and Remarks

Copyright and Distribution

© 2023 by Trail of Bits, Inc.

All rights reserved. Trail of Bits hereby asserts its right to be identified as the creator of this report in the United Kingdom.

This report is considered by Trail of Bits to be public information; it is licensed to Salty.IO under the terms of the project statement of work and has been made public at Salty.IO's request. Material within this report may not be reproduced or distributed in part or in whole without the express written permission of Trail of Bits.

The sole canonical source for Trail of Bits publications is the Trail of Bits Publications page. Reports accessed through any source other than that page may have been modified and should not be considered authentic.

Test Coverage Disclaimer

All activities undertaken by Trail of Bits in association with this project were performed in accordance with a statement of work and agreed upon project plan.

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

Trail of Bits uses automated testing techniques to rapidly test the controls and security properties of software. These techniques augment our manual security review work, but each has its limitations: for example, a tool may not generate a random edge case that violates a property or may not fully complete its analysis during the allotted time. Their use is also limited by the time and resource constraints of a project.

Table of Contents

Executive Summary

Engagement Overview

Salty.IO engaged Trail of Bits to review the security of the Salty.IO protocol.

A team of two consultants conducted the review from October 16 to November 3, 2023, for a total of six engineer-weeks of effort. Our testing efforts focused on the pools, collateral, liquidity, and stablecoin components. With full access to the source code and documentation, we performed static analysis using automated and manual processes.

Observations and Impact

Salty.IO is a DeFi protocol that incorporates innovative features, including arbitrage and counterswaps. The novelty of the protocol introduces intrinsic risks, as the new patterns have not been stress tested over time. Furthermore, due to its nascent stage, the codebase's complexity management, testing, documentation, and adherence to best practices could benefit from further refinement.

We identified several high-severity issues during the review. Some findings, such as TOB-SALTY-5, TOB-SALTY-7, and TOB-SALTY-8, stem in part from the complexity of the inheritance structure. Other issues, like TOB-SALTY-11 and TOB-SALTY-12, stem from the system's reliance on manual processes that require timeliness. Finally, as described in TOB-SALTY-12, there are questions concerning the protocol's resilience and the collateralization of the USDS stablecoin, especially in the context of significant market anomalies like flash crashes.

Recommendations

Based on the codebase maturity evaluation and findings identified during the security review, Trail of Bits recommends that Salty.IO take the following steps:

- **Remediate the findings disclosed in this report.** These findings should be addressed as part of a direct remediation or as part of any refactor that may occur when addressing other recommendations.
- **Incorporate Slither into the CI pipeline.** Slither can help the development team proactively address informational-level concerns and align the protocol with best practices.
- **Implement a fuzzing campaign.** Create a suite of fuzz tests to test function properties, arithmetic, and system invariants such as pool ratios.
- **Have a targeted review performed on the areas of the protocol that were not fully covered during this audit.** These areas include those that were out of scope

or received limited review due to time constraints, including `Upkeep`, `Deployment`, `Utils`, `Airdrop`, `Bootstrap`, `InitialDistribution`, DAO, and related contracts. Refer to the Project Coverage section for more information on system elements that may warrant further review.

- **Have an additional review performed if significant changes are made to the protocol.** For example, if logic is moved off-chain, the liquidation mechanisms are changed, or any other major architectural changes are implemented, consider conducting a new review of the overall protocol.

The following tables provide the number of findings by severity and category.

EXPOSURE ANALYSIS

Severity	*Count*
High	6
Medium	2
Low	2
Informational	3
Undetermined	0

CATEGORY BREAKDOWN

Category	*Count*
Auditing and Logging	1
Authentication	2
Configuration	1
Data Validation	8
Denial of Service	1

Project Summary

Contact Information

The following project manager was associated with this project:

Sam Greenup, Project Manager
sam.greenup@trailofbits.com

The following engineering director was associated with this project:

Josselin Feist, Engineering Director, Blockchain
josselin.feist@trailofbits.com

The following engineers were associated with this project:

Richie Humphrey, Consultant
Richie.Humphrey@trailofbits.com

Damilola Edwards, Consultant
Damilola.Edwards@trailofbits.com

Project Timeline

The significant events and milestones of the project are listed below.

Date	Event
October 12, 2023	Pre-project kickoff call
October 23, 2023	Status update meeting #1
October 30, 2023	Status update meeting #2
November 3, 2023	Delivery of report draft
November 6, 2023	Report readout meeting
December 13, 2023	Delivery of final report with fix review appendix

Project Goals

The engagement was scoped to provide a security assessment of the Salty.IO protocol. Specifically, we sought to answer the following non-exhaustive list of questions:

- Can pools be drained by an attacker?
- Could funds be lost, through either accident or attacks?
- Can functionality be disabled wholly or partially?
- Are there front-running vulnerabilities in the system?
- Are there appropriate access controls in place?
- Is it possible to perform swaps without paying the required amount?

Project Targets

The engagement involved a review and testing of the following target.

Salty.IO

Repository	othernet-global/salty-io
Version	`9dbd6c0c963ff5cfb177e106a5c470a39fb0bb5f`
Type	Solidity
Platform	EVM

The review involved the Solidity contracts in the following folders, excluding subfolders, tests, and interfaces:

- `arbitrage/`
- `dao/`
- `launch/`
- `pools/`
- `price_feed/`
- `rewards/`
- `stable/`
- `staking/`
- `./` (root)

Project Coverage

This section provides an overview of the analysis coverage of the review, as determined by our high-level engagement goals. Our approaches are detailed below.

Aside from the exceptions noted in the Coverage Limitations section, we performed the following actions for all in-scope contracts:

- Conducted a manual line-by-line analysis
- Searched for issues that would result in the loss of funds
- Reviewed the implementation of access controls
- Ran Slither to detect vulnerabilities and violations of smart contract best practices
- Created diagrams and charts to better understand the key contracts and workflows

For each section of the code, we used customized approaches to explore pertinent attack vectors and potential problems as follows:

Pools

- Independently verified the math for swapping and zapping
- Looked for ways the pool could be drained, including front-running and flash-loan attacks
- Considered ways in which the counterswap feature could be manipulated
- Created a model of the arbitrage feature and examined multiple scenarios

Liquidity

- Independently verified the math for adding/removing liquidity
- Looked for issues with the initial deposit
- Sought ways to game the liquidity reward system
- Reviewed the slippage mechanisms

Collateral/Lending/Stablecoin

- Conducted a manual review to check for possible ways in which bad debt could be introduced into the system and the mitigation strategies in place
- Reviewed the collateralization and liquidation mechanisms

- Reviewed how external data sources (oracles) are integrated for price feeds
- Determined whether proper access control mechanisms, roles, and permissions are enforced to prevent unauthorized actions
- Reviewed the liquidation process for undercollateralized loans
- Reviewed the economic incentives to understand how the protocol rewards various actors and maintains stability
- Reviewed the Chainlink and Uniswap documentation to ensure that the integrations with these protocols are in line with best practices

Staking/Rewards

- Reviewed the logic for staking tokens, including locking periods and penalty calculations for early withdrawals
- Examined the manual steps required to maintain the staking mechanism and looked for any potential pitfalls
- Reviewed the reward distribution mechanism, including the calculation of rewards and distribution intervals
- Reviewed the governance and voting rights associated with staked tokens

Coverage Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. The following list outlines the coverage limitations of the engagement and indicates system elements that may warrant further review:

- The following system elements were out of scope and, therefore, were not reviewed at all: the `Upkeep`, `Deployment`, and `Utils` contracts, the system monitoring mechanism and incident response plan, and the off-chain components.
- We reviewed the following contracts, but due to time constraints, we could not review them thoroughly: `DAO`, `DAOConfig`, `Parameters`, `Proposals`, `Airdrop`, `Bootstrap`, and `InitialDistribution`.
- Due to time constraints, we did not conduct a fuzzing campaign to test function properties and system invariants. Specifically, we would have liked to fuzz the following areas:
    - `Pools` contract invariants
    - `Liquidity/Collateral` contract invariants

    - Stablecoin/lending invariants
    - All math and math-related functions, including those for swapping, zapping, and adding/removing liquidity

- Due to time constraints, we did not spend as much time as we would have liked inspecting the arithmetic for precision and rounding-related issues. These types of issues may appear to be minor, but they can compound into high-severity bugs. Fuzzing can also be used to identify edge cases and other problems.
- Per the client's instructions, we did not spend significant time looking for and reporting on findings related to best practices or other informational-level findings.
- Due to time constraints, we did not review the entire test suite for accuracy or completeness.

Codebase Maturity Evaluation

Trail of Bits uses a traffic-light protocol to provide each client with a clear understanding of the areas in which its codebase is mature, immature, or underdeveloped. Deficiencies identified here often stem from root causes within the software development life cycle that should be addressed through standardization measures (e.g., the use of common libraries, functions, or frameworks) or training and awareness programs.

Category	Summary	Result
Arithmetic	In general, the arithmetic used in the protocol is documented and implemented with an emphasis on clarity and understandability. The code does not use any unchecked blocks. However, some arithmetic formulas do not have specifications, and some have specifications that do not match the code. We did not identify an explicit testing strategy to increase confidence in the system's arithmetic. The testing does not cover several arithmetic edge cases, including critical ones. While there are some comments around precision and rounding issues, there is room for improvement. There are areas where unsafe casting is performed. This should be addressed either by changing the code or adding a comment justifying the decision not to use safe casting.	**Moderate**
Auditing	The protocol does not use any events and, therefore, lacks mechanisms for monitoring system behavior and tracking critical component updates. Without event-emitting functions, off-chain monitoring detection of unexpected system behavior is hindered, and integration with front-end systems is complicated.	**Weak**
Authentication / Access Controls	Access controls are generally satisfactory, but we did identify two findings where access controls were missing due to an oversight during the configuration of the contracts (TOB-SALTY-8 and TOB-SALTY-9).	**Moderate**

Complexity Management	Though certain areas of complexity are well documented and implemented in a sustainable way, we noted several areas of excess complexity. For example, there are numerous contracts in the protocol; instances in which a function makes an external call just to fetch an address for yet another external call add undue complexity and offer opportunities for streamlining. Additionally, some functions have unclear scopes, or their scopes include too many components. Furthermore, as noted in TOB-SALTY-5, TOB-SALTY-7, and TOB-SALTY-8, there is complexity associated with inheritance, some of which can be addressed through the use of abstract contracts.	**Weak**
Decentralization	The protocol has many decentralized aspects, and decentralization is embedded in the design. For example, unlike other DAO proposals that often allow for the execution of arbitrary calldata, Salty.IO limits the actions available to the DAO. Furthermore, we found functions in which privileged roles could be abused. However, this protocol is not permissionless. All users must be whitelisted to do anything other than swapping. Furthermore, if a new country is added to the blacklist, every user must be whitelisted again in order to add liquidity, take loans, or stake.	**Moderate**
Documentation	The Salty.IO team has created some helpful external user and technical documentation. The codebase does not contain any NatSpec comments, as recommended by the Solidity style guide. However, the codebase has robust inline code comments throughout and includes several areas of well-commented code.	**Satisfactory**
Transaction Ordering	Transaction ordering risks have been considered in the design of Salty.IO, but there are still risks and areas whose resistance against such risks is unknown. The fact that there are no fees on swaps invites pool	**Moderate**

	manipulation by flash loans or whales. Findings TOB-SALTY-3 and TOB-SALTY-6 are related to this. The concept of arbitrage appears to be an effective way of mitigating the risks of sandwiching transactions with swaps to manipulate the reserve ratio. However, that mitigation is not guaranteed to be 100% effective due to multiple factors such as the size and ratio of the reserves of the intermediary pools. For swaps, the slippage guards in place are an effective mitigation against front-running. Regardless of the decision to mitigate them, all known risks related to transaction ordering should be clearly documented.	
Low-Level Manipulation	The code does not use any low-level manipulation.	**Satisfactory**
Testing and Verification	There is a large suite of unit and integration tests that interacts with a fork of the Sepolia testnet. The tests appear to cover most common use cases of the protocol and test a fair number of potential reverts or other scenarios outside of the “happy path.” However, there is no targeted fuzz testing of arithmetic operations, invariants, or function properties. Furthermore, there is no mutation testing. These methodologies can expose unforeseen edge cases or anomalies that regular testing might miss. Fuzzing involves testing with random data inputs to trigger unhandled exceptions or crashes, while mutation testing, a method of code quality validation, alters the software code in small ways to assess whether the test cases can distinguish the original code from the mutated one. These can help ensure the resistance of the application against potential unusual inputs or behaviors.	**Moderate**

Summary of Findings

The table below summarizes the findings of the review, including type and severity details.

ID	Title	Type	Severity
7	Collateral contract deployment results in permanent loss of rewards	Configuration	High
8	Collateral can be withdrawn without repaying USDS loan	Authentication	High
11	Lack of timely price feed updates may result in loss of funds	Data Validation	High
1	Risk of denial-of-service attacks on token whitelisting process	Denial of Service	High
3	Transactions to add liquidity may be front run	Data Validation	High
12	USDS stablecoin may become undercollateralized	Data Validation	High
5	Any user can add liquidity to any pool and bypass the token whitelist	Data Validation	Medium
6	Liquidation fee is volatile and may be manipulated	Data Validation	Medium
13	Zap operations may approve an incorrect number of tokens, leading to reversion	Data Validation	Low
4	Whitelisted pools may exceed the maximum allowed	Data Validation	Low
2	Insufficient event generation	Auditing and Logging	Informational

9	Lack of chain ID validation allows signature reuse across forks	Authentication	**Informational**
10	Chainlink oracles could return stale price data	Data Validation	**Informational**

Detailed Findings

1. Risk of denial-of-service attacks on token whitelisting process

Severity: **High**	Difficulty: **Medium**
Type: Denial of Service	Finding ID: TOB-SALTY-1
Target: `src/dao/Proposal.sol`	

Description

The token whitelisting process can be disrupted by a malicious user submitting spurious proposals.

The `proposeTokenWhitelisting` function allows users to submit proposals for adding new tokens to the whitelist. If the requirements are met, it creates a unique proposal and includes it in the pending whitelisting proposals list.

```
 121    function proposeTokenWhitelisting( IERC20 token, string memory tokenIconURL,
string memory description ) public nonReentrant
 122         {
 123         require( address(token) != address(0), "token cannot be address(0)" );
 124
 125         require( _openBallotsForTokenWhitelisting.length() <
daoConfig.maxPendingTokensForWhitelisting(), "The maximum number of token
whitelisting proposals are already pending" );
 126         require( poolsConfig.numberOfWhitelistedPools() <
poolsConfig.maximumWhitelistedPools(), "Maximum number of whitelisted pools already
reached" );
 127         require( ! poolsConfig.tokenHasBeenWhitelisted(token,
exchangeConfig.wbtc(), exchangeConfig.weth()), "The token has already been
whitelisted" );
 128
 129         string memory ballotName = string.concat("whitelist:",
Strings.toHexString(address(token)) );
 130
 131         uint256 ballotID = _possiblyCreateProposal( ballotName,
BallotType.WHITELIST_TOKEN, address(token), 0, tokenIconURL, description, 2 *
daoConfig.baseProposalCost() );
 132         _openBallotsForTokenWhitelisting.add( ballotID );
 133         }
```

Figure 1.1: src/dao/Proposals.sol#L121-L133

However, the maximum number of queueable token whitelisting proposals is currently capped at five, which can be extended to a maximum of 12. This presents a potential vulnerability to denial-of-service attacks, as a malicious actor could persistently submit meaningless tokens to saturate the queue, thereby preventing legitimate token whitelisting proposals.

```
 49    // The maximum number of tokens that can be pending for whitelisting at any
time.
 50    // Range: 3 to 12 with an adjustment of 1
 51    uint256 public maxPendingTokensForWhitelisting = 5;
```

Figure 1.2: `src/dao/DAOConfig.sol#L49-L51`

Furthermore, since voters may lack incentive to downvote these fraudulent proposals, the required quorum may never be reached to finalize the ballot, effectively blocking the service for an extended duration.

Submitting a proposal requires a nonrefundable fee that is currently set at $500, which should prevent most frivolous attackers. However, this will not stop a motivated, well-funded attacker such as a competitor.

Exploit Scenario

When the Salty.IO protocol is deployed, Eve submits multiple proposals to whitelist fake tokens. These proposals fill up the queue, effectively obstructing the inclusion of genuine token whitelisting proposals in the queue. Once the protocol is able to get a quorum to remove one of the proposals, Eve's bot immediately back runs the transaction with another proposal.

Recommendations

Short term, implement a mechanism that allows an authorized entity to remove undesirable or unwanted token whitelisting proposals from the queue of proposals. Alternatively, implement a larger sized deposit that would be returned to a legitimate proposer.

Long term, review critical codebase operations to ensure consistent and comprehensive logging of essential information, promoting transparency, troubleshooting, and efficient system management.

2. Insufficient event generation	
Severity: **Informational**	Difficulty: **Low**
Type: Auditing and Logging	Finding ID: TOB-SALTY-2
Target: `src/*`	

Description
The protocol does not use events at all. As a result, it will be difficult to review the contracts' behavior for correctness once they have been deployed.

Events generated during contract execution aid in monitoring, baselining of behavior, and detection of suspicious activity. Without events, users and blockchain-monitoring systems cannot easily detect behavior that falls outside the baseline conditions; malfunctioning contracts and attacks could go undetected.

Exploit Scenario
An attacker discovers a vulnerability in one of the Salty.IO protocol contracts and modifies its execution. Because these actions generate no events, the behavior goes unnoticed until there is follow-on damage, such as financial loss.

Recommendations
Short term, add events for all operations that could contribute to a higher level of monitoring and alerting.

Long term, consider using a blockchain-monitoring system to track any suspicious behavior in the contracts. The system relies on several contracts to behave as expected. A monitoring mechanism for critical events would quickly detect any compromised system components.

3. Transactions to add liquidity may be front run

Severity: **High**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-SALTY-3
Target: `src/pools/Pools.sol`	

Description

Even with a properly set slippage guard, a depositor calling `addLiquidity` can be front run by a sole liquidity holder, which would cause the depositor to lose funds.

When either of the reserves of a pool is less than the dust (100 wei), any call to `addLiquidity` will cause the pool to reset the reserves ratio.

```
// If either reserve is less than dust then consider the pool to be empty and that
the added liquidity will become the initial token ratio
if ( ( reserve0 <= PoolUtils.DUST ) || ( reserve1 <= PoolUtils.DUST ) )
       {
       // Update the reserves
       reserves.reserve0 += maxAmountA;
       reserves.reserve1 += maxAmountB;

       return ( maxAmountA, maxAmountB, Math.sqrt(maxAmountA * maxAmountB) );
       }
```

Figure 3.1: `src/pools/Pools.sol#L112-L120`

Exploit Scenario

Eve is the sole liquidity holder in the WETH/DAI pool, which has reserves of 100 WETH and 200,000 DAI, for a ratio of 1:2,000.

1. Alice submits a transaction to add 10 WETH and 20,000 DAI of liquidity.
2. Eve front runs Alice's transaction with a `removeLiquidity` transaction that brings one of the reserves down to close to zero.
3. As the last action of her transaction, Eve adds liquidity, but because one of the reserves is close to zero, whatever ratio she adds to the pool becomes the new reserves ratio. In this example, Eve adds the ratio 10:2,000 (representing a WETH price of 200 DAI).
4. Alice's `addLiquidity` transaction goes through, but because of the new K ratio, the logic lets her add only 10 WETH and 2,000 DAI. The liquidity slippage guard does not

work because the reserves ratio has been reset. In nominal terms, Alice actually receives more liquidity than she would have at the previous ratio.

5. Eve back runs this transaction with a swap transaction that buys most of the WETH that Alice just deposited for a starting price of 200 DAI. Eve then removes her liquidity, effectively stealing Alice's 10 WETH for a fraction of the price.

Recommendations

Short term, add a mechanism to prevent reserves from dropping below dust levels.

Long term, carefully assess invariants and situations in which front-running may affect functionality. Invariant testing would be useful for this.

4. Whitelisted pools may exceed the maximum allowed	
Severity: **Low**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-SALTY-4
Target: `src/pools/PoolsConfig.sol`	

Description
The number of whitelisted pools may exceed the maximum number allowed because there is no check against the current length of `whitelistedPools` when the maximum number of `whitelistedPools` is decreased via a proposal.

```
function changeMaximumWhitelistedPools(bool increase) public onlyOwner
 {
 if (increase)
     {
     if (maximumWhitelistedPools < 100)
         maximumWhitelistedPools += 10;
     }
 else
     {
     if (maximumWhitelistedPools > 20)
         maximumWhitelistedPools -= 10;
     }
 }
```

Figure 4.1: `src/pools/PoolsConfig.sol#L59-L72`

Exploit Scenario
The current `maximumWhitelistedPools` is set to `20`, and the current number of whitelisted pools is 20. A proposal that decreases the maximum number of pools to 10 is approved, so `maximumWhitelistedPools` becomes `10`. There are three other proposals to whitelist new tokens, which all pass, creating six new whitelisted pools. This brings the total number of whitelisted pools to 26, while the maximum is 10. In order to add a new whitelisted pool, two new proposals must pass that each increase the maximum by 10.

Recommendations
Short term, add a check of the current number of pools and the number of proposals for adding new tokens; the check should run before the maximum is changed and before proposals to whitelist tokens are accepted.

Long term, design a specification that identifies the boundaries for all parameters along with requirements as to how situations such as these should be handled.

5. Any user can add liquidity to any pool and bypass the token whitelist	
Severity: **Medium**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-SALTY-5
Target: `src/pools/Pools.sol`	

Description

The absence of token whitelist checks within the `addLiquidity` function allows any user to create trading pairs, including pairs with malicious tokens, and to provide liquidity for arbitrary token pairs. This vulnerability may expose the protocol to potential reentrancy attacks, thereby resulting in the potential loss of funds for users who provide liquidity for the pool. Furthermore, the lack of authorization checks on this function allows users from blacklisted regions to directly call `addLiquidity` on a pool, bypassing the `AccessManager` access control and associated geolocation check.

```
// Add liquidity for the specified trading pair (must be whitelisted)
function addLiquidity( IERC20 tokenA, IERC20 tokenB, uint256 maxAmountA, uint256
maxAmountB, uint256 minLiquidityReceived, uint256 deadline ) public nonReentrant
ensureNotExpired(deadline) returns (uint256 addedAmountA, uint256 addedAmountB,
uint256 addedLiquidity)
	{
	require(
exchangeConfig.initialDistribution().bootstrapBallot().startExchangeApproved(), "The
exchange is not yet live" );
	require( address(tokenA) != address(tokenB), "Cannot add liquidity for
duplicate tokens" );
```

Figure 5.1: The `addLiquidity` function for the `Liquidity` contract (`src/pools/Pools.sol#L149-L177`)

Exploit Scenario

Alice creates a trading pair using a malicious token. The malicious token pair allows attackers to perform reentrancy attacks and to steal funds from Alice, who provides liquidity in the associated pool.

Recommendations

Short term, add the appropriate token whitelist and user authorization checks to the `addLiquidity` function.

Long term, review critical operations in the codebase and ensure that proper access control mechanisms are put in place.

6. Liquidation fee is volatile and may be manipulated

Severity: **Medium**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-SALTY-6
Target: `src/stable/Collateral.sol`	

Description

The liquidation reward is intended to be 5% of the liquidity seized, but in actuality, it may range from close to 0% to close to 10%.

The Salty.IO liquidation process is straightforward. When the loan-to-value (LTV) ratio of a loan drops below the threshold, `liquidateUser` may be called by anyone, and it will immediately liquidate the loan by removing the liquidity, seizing the WBTC and WETH, and paying a fee to the caller.

The fee is intended to be 5% of the amount liquidated, which is estimated as 10% of the WETH seized. This assumes the WETH and WBTC will be equal in value. However, due to market forces, the actual ratio of values will diverge from 50:50 even if that was the ratio of the values of the initial deposit.

```
// Liquidate a position which has fallen under the minimum collateral ratio.
// A default 5% of the value of the collateral is sent to the caller, with the rest
being sent to the Liquidator for later conversion to USDS which is then burned.
function liquidateUser( address wallet ) public nonReentrant

        …

        // The caller receives a default 5% of the value of the liquidated collateral
so we can just send them default 10% of the reclaimedWETH (as WBTC/WETH is a 50/50
pool).
        uint256 rewardedWETH = (2 * reclaimedWETH *
stableConfig.rewardPercentForCallingLiquidation()) / 100;

        …

        // Reward the caller
        weth.safeTransfer( msg.sender, rewardedWETH );
```

Figure 6.1: `src/stable/Collateral.sol#L144-L184`

The value of WETH can be further manipulated by sandwiching a call to `liquidateUser` with a WBTC/WETH swap transaction, which could push the reserves of either token to

close to the dust amount. This could double the amount of WETH received by the liquidator; alternatively, it could reduce the amount of WETH to almost zero.

Users who want to add liquidity, called LPs, and participate in LP staking rewards are expected to call the `Liquidity` contract; however, a user could also call `addLiquidity` directly on a pool, bypassing the `AccessManager` access control and associated geolocation check. Furthermore, a user may believe that by doing so they will participate in the arbitrage profit rewards, but in fact, they will not.

Exploit Scenario

Alice takes out a loan of USDS against her WBTC/WETH.

Subsequently, Eve notices the LTV of the loan is below the threshold. She submits a transaction:

1. It first swaps WETH for a significant amount of WBTC from the collateral pool.
2. It then calls `liquidateUser`, and due to the estimate based on WETH, Alice receives significantly more WETH than she was entitled to.
3. Finally, it unwinds the swap from step 1.

Recommendations

Short term, instead of estimating the fee as two times the WETH amount, have the `liquidateUser` function send the exact amount of WETH and WBTC to the liquidator.

Long term, create a specification for all major workflows with clear definitions and expectations for each process.

7. Collateral contract deployment results in permanent loss of rewards

Severity: **High**	Difficulty: **Low**
Type: Configuration	Finding ID: TOB-SALTY-7
Target: `src/rewards/RewardsEmitter.sol`	

Description

Anyone who provides liquidity to the collateral pool (also known as the WETH/WBTC pool) will not earn rewards, and any arbitrage profits allocated to the pool will be permanently lost.

The `Collateral` contract is deployed separately from the `Liquidity` contract; however, only liquidity in the `Liquidity` contract can claim rewards.

This arises because the `Liquidity` contract is the sole contract seeded with SALT rewards. As shown in figure 7.1, the `performUpkeep` function, which is responsible for sending rewards to the relevant contract, references only `stakingRewards`, which is configured as the address of the `Liquidity` contract; it completely overlooks the `Collateral` contract. This means that rewards intended for the collateral pool are misdirected, as they should properly be sent to the `Collateral` contract.

```
	// Transfer a percent (default 1% per day) of the currently held rewards to
the specified StakingRewards pools.
	// The percentage to transfer is interpolated from how long it's been since
the last performUpkeep().
	function performUpkeep( uint256 timeSinceLastUpkeep, bool isStaking ) public

		…

		// Add the rewards so that they can later be claimed by the users
proportional to their share of the StakingRewards derived contract( Staking,
Liquidity or Collateral)
		stakingRewards.addSALTRewards( addedRewards );
		}
```

Figure 7.1: In the above code, `stakingRewards` refers to the `Liquidity` contract. (`src/rewards/RewardsEmitter.sol#L80-L138`)

If an actual collateral pool liquidity holder calls `claimAllRewards` on the `Collateral` contract, then it will revert because there are no rewards in the contract. If the user calls

`claimAllRewards` on the `Liquidity` contract, it will revert because the user has no liquidity as far as the `Liquidity` contract is concerned.

Exploit Scenario
The protocol is deployed, and Alice adds liquidity to the collateral pool. Over time, the pool accumulates arbitrage fees. Alice is unable to claim any rewards, and the rewards that are sent to the pool are permanently lost.

Recommendations
Short term, combine the `Liquidity` and `Collateral` contracts into one.

Long term, create a specification for all contracts with clear definitions and expectations for each process.

8. Collateral can be withdrawn without repaying USDS loan

Severity: **High**	Difficulty: **Low**
Type: Authentication	Finding ID: TOB-SALTY-8
Target: `src/stable/Collateral.sol`	

Description

Users could bypass collateralization checks and reclaim their collateral assets without repaying their USDS loans.

The `withdrawCollateralAndClaim` function withdraws WBTC/WETH collateral assets and sends any pending rewards to the caller. It includes checks to ensure that, after the withdrawal, the caller will still possess sufficient collateral in the form of liquidity shares to maintain any outstanding loans before initiating the withdrawal of WBTC and WETH liquidity from the pool and sending the reclaimed tokens back to the user.

```
 81      function withdrawCollateralAndClaim( uint256 collateralToWithdraw, uint256
minReclaimedWBTC, uint256 minReclaimedWETH, uint256 deadline ) public returns
(uint256 reclaimedWBTC, uint256 reclaimedWETH)
 82                {
 83                // Make sure that the user has collateral and if they have
borrowed USDS that collateralToWithdraw doesn't bring their collateralRatio below
allowable levels.
 84                require( userShareForPool( msg.sender, collateralPoolID ) > 0,
"User does not have any collateral" );
 85                require( collateralToWithdraw <=
maxWithdrawableCollateral(msg.sender), "Excessive collateralToWithdraw" );
 86
 87                // Withdraw the WBTC/WETH liquidity from the liquidity pool
(sending the reclaimed tokens back to the user)
 88                (reclaimedWBTC, reclaimedWETH) = withdrawLiquidityAndClaim(
wbtc, weth, collateralToWithdraw, minReclaimedWBTC, minReclaimedWETH, deadline );
 89                }
```

Figure 8.1: `src/stable/Collateral.sol#L81-L88`

However, the `withdrawLiquidityAndClaim` function, which is called by `withdrawCollateralAndClaim`, has public visibility with no access control mechanism in place. This means that the collateralization checks within the `withdrawCollateralAndClaim` function can be bypassed, and users can directly invoke `withdrawLiquidityAndClaim` to reclaim their liquidity assets while holding onto their USDS loans.

```
 77        function withdrawLiquidityAndClaim( IERC20 tokenA, IERC20 tokenB, uint256
liquidityToWithdraw, uint256 minReclaimedA, uint256 minReclaimedB, uint256 deadline
) public nonReentrant returns (uint256 reclaimedA, uint256 reclaimedB)
 78                {
 79                // Make sure that the DAO isn't trying to remove liquidity
 80                require( msg.sender != address(exchangeConfig.dao()), "DAO is
not allowed to withdraw liquidity" );
 81
 82                (bytes32 poolID,) = PoolUtils._poolID( tokenA, tokenB );
 83
 84                // Reduce the user's liquidity share for the specified pool so
that they receive less rewards.
 85                // Cooldown is specified to prevent reward hunting (ie - quickly
depositing and withdrawing large amounts of liquidity to snipe rewards)
 86                // This call will send any pending SALT rewards to msg.sender as
well.
 87                // Note: _decreaseUserShare checks to make sure that the user
has the specified liquidity share.
 88                _decreaseUserShare( msg.sender, poolID, liquidityToWithdraw,
true );
 89
 90                // Remove the amount of liquidity specified by the user.
 91                // The liquidity in the pool is currently owned by this
contract. (external call)
 92                (reclaimedA, reclaimedB) = pools.removeLiquidity( tokenA,
tokenB, liquidityToWithdraw, minReclaimedA, minReclaimedB, deadline );
 93
 94                // Transfer the reclaimed tokens to the user
 95                tokenA.safeTransfer( msg.sender, reclaimedA );
 96                tokenB.safeTransfer( msg.sender, reclaimedB );
 97                }
 98          }
```

Figure 8.2: src/staking/Liquidity.sol#L77-L98

Exploit Scenario

Alice borrows USDS tokens from the protocol using her liquidity shares as collateral. Instead of repaying the loan to reclaim her liquidity shares, she directly accesses the `withdrawLiquidityAndClaim` function. By doing so, she withdraws her collateral assets while still holding onto her USDS loan, thereby stealing funds from the protocol.

Recommendations

Short term, restrict direct public access to the `withdrawLiquidityAndClaim` function. Consider changing its visibility to “internal” or introducing access control checks.

Long term, review critical operations in the codebase and ensure that proper access control mechanisms are put in place.

9. Lack of chain ID validation allows signature reuse across forks

Severity: **Informational**	Difficulty: **High**
Type: Authentication	Finding ID: TOB-SALTY-9
Target: `src/accessManager.sol`	

Description

Without chain ID validation, a valid signature obtained on one blockchain fork or network can be reused on another with a different chain ID.

The `grantAccess` function is responsible for validating a provided signature and granting access to a user within the context of a specific geographical version (`geoVersion`). It verifies the authenticity of the signature, ensuring that it was generated by an authorized source, typically an off-chain verifier. Upon successful verification, the user's wallet status is upgraded and granted access.

```
 56          // Grant access to the sender for the given geoVersion (which is
incremented when new regions are restricted).
 57          // Requires the accompanying correct message signature from the off
chain verifier.
 58      function grantAccess(bytes memory signature) public
 59          {
 60          require( verifyWhitelist(msg.sender, signature), "Incorrect
AccessManager.grantAccess signatory" );
 61
 62           _walletsWithAccess[geoVersion][msg.sender] = true;
 63          }
```

Figure 9.1: The `grantAccess` function

However, the signature schema does not account for the contract's chain. As a result, if the protocol is deployed on multiple chains, one valid signature will be reusable across all of the available forks.

Recommendations

Short term, add the chain ID opcode to the signature schema.

Long term, identify and document the risks associated with having forks of multiple chains and develop related mitigation strategies.

10. Chainlink oracles could return stale price data

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-SALTY-10
Target: `src/price_feed/CoreChainlinkFeed.sol`	

Description

The `latestRoundData()` function from Chainlink oracles returns five values: `roundId`, `answer`, `startedAt`, `updatedAt`, and `answeredInRound`. The `PriceConverter` contract reads only the `answer` value and discards the rest. This can cause outdated prices to be used for token conversions.

```
 27    // Returns a Chainlink oracle price with 18 decimals (converted from
Chainlink's 8 decimals).
 28    // Returns zero on any type of failure.
 29    function latestChainlinkPrice(address _chainlinkFeed) public view returns
(uint256)
 30          {
 31          AggregatorV3Interface chainlinkFeed =
AggregatorV3Interface(_chainlinkFeed);
 32
 33          int256 price = 0;
 34
 35          try chainlinkFeed.latestRoundData()
 36          returns (
 37                uint80, // _roundID
 38                int256 _price,
 39                uint256, // _startedAt
 40                uint256, // _timeStamp
 41                uint80 // _answeredInRound
 42          )
 43                {
 44                price = _price;
 45                }
```

Figure 10.1: All returned data other than the `answer` value is ignored during the call to a Chainlink feed's `latestRoundData` method. (`src/price_feed/CoreChainlinkFeed.sol#L67–L71`)

According to the Chainlink documentation, if the `latestRoundData()` function is used, the `updatedAt` value should be checked to ensure that the returned value is recent enough for the application.

Recommendations

Short term, make sure that the oracle queries check for up-to-date data. In the case of stale oracle data, have the `latestRoundData()` function return zero to indicate failure.

Long term, review the documentation for Chainlink and other oracle integrations to ensure that all of the security requirements are met to avoid potential issues, and add tests that take these possible situations into account.

11. Lack of timely price feed updates may result in loss of funds

Severity: **High**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-SALTY-11
Target: `src/price_feed/PriceAggregator.sol`	

Description

The prices used to check loan health and maximum borrowable value are updated manually by calling `performUpkeep` on the `PriceAggregator` contract. There are incentives to call the price update function, and the client has indicated it will be running a bot to perform this and other upkeeps, but there is no guarantee that this will happen.

The most recently updated prices are stored in the `_lastPriceOnUpkeepBTC` and `_lastPriceOnUpkeepETH` lists; however, there is no indication as to when they were last updated.

```
 function _updatePriceBTC() internal
       {
       uint256 price1 = _getPriceBTC(priceFeed1);
       uint256 price2 = _getPriceBTC(priceFeed2);
       uint256 price3 = _getPriceBTC(priceFeed3);

       _lastPriceOnUpkeepBTC = _aggregatePrices(price1, price2, price3);
       }


function _updatePriceETH() internal
       {
       uint256 price1 = _getPriceETH(priceFeed1);
       uint256 price2 = _getPriceETH(priceFeed2);
       uint256 price3 = _getPriceETH(priceFeed3);

       _lastPriceOnUpkeepETH = _aggregatePrices(price1, price2, price3);
       }


// Cache the current prices of BTC and ETH until the next performUpkeep
// Publicly callable without restriction as the function simply updates the BTC and
ETH prices as read from the PriceFeed (which is a trusted source of price).
function performUpkeep() public
       {
       _updatePriceBTC();
       _updatePriceETH();
       }
```

Figure 11.1: src/stable/Collateral.sol#L81-L88

Exploit Scenario

The Salty.IO bot goes down and does not update prices. During this time, the market drops sharply. Because the prices are stale, Eve is able to borrow significantly at inflated prices. When the bot finally resumes, all of Eve's loans are below 100% collateralization.

Recommendations

Short term, have the code store the timestamp of the last update and have the price checks revert if the timestamp is too old.

Long term, remove the price caching pattern altogether and replace it with a check of the price feeds when `getPriceBTC` or `getPriceETH` are called.

12. USDS stablecoin may become undercollateralized	
Severity: **High**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-SALTY-12
Target: `src/stable/USDS.sol`	

Description
In a downtrending market, a black swan type event could cause the USDS stablecoin to become undercollateralized.

When a loan is liquidated, 5% of the collateral is sent to the liquidator as a fee. If 105% of the collateral is not received, the protocol will suffer a loss.

Furthermore, the collateral seized from liquidations is not immediately sold; instead, it is registered as a counterswap to be sold the next time a matching buy order is placed. In a downtrending market, especially during a flash crash, this forces the protocol to retain these assets as they decline in value.

We also noted that the collateralization ratio is calculated assuming that the price of USDS is $1.

Exploit Scenario
The market crashes, and liquidations are triggered. The liquidations result in the seizure of WBTC and WETH, which are deposited in the `Pools` contract until a counterswap occurs. As the market continues to fall, no buy orders are placed, and no counterswaps are triggered. Salty.IO suffers losses from being naked long the tokens. When the flash crash bottoms out and reverses, the counterswaps are immediately triggered, locking in the unrealized losses at the bottom.

Recommendations
Short term, analyze the issue and identify a possible solution for this problem. This problem does not have a simple solution and is inherent in the design of the protocol. Any solution should be thoughtfully considered and tested. Risks that are not addressed should be clearly documented.

Long term, create a design specification that identifies the interactions and risks between the protocol's features. Test scenarios combined with fuzzing can be used to simulate adverse market conditions and identify weaknesses.

13. Zap operations may approve an incorrect number of tokens, leading to reversion

Severity: **Low**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-SALTY-13
Target: `src/staking/Liquidity.sol`	

Description

During the audit, the client found and reported this issue, stating the following:

> *I noticed an issue that prevents zapping from working correctly.*
>
> *`Liquidity.addLiquidityAndIncreaseShare` approves Pool spending of the tokens before the zapping is done, but zapping can adjust the deposited amounts above the initial maxAmounts, in which case the `pools.addLiquidity` call will fail with insufficient allowance.*
>
> *I'll move the approval after the zap so the correct amount is approved.*
>
> *The unit tests did not notice it, as they approved `unit256.max` and dealt with `Pools.dualZapInLiquidity` directly.*

```
function addLiquidityAndIncreaseShare( IERC20 tokenA, IERC20 tokenB, uint256
maxAmountA, uint256 maxAmountB, uint256 minLiquidityReceived, uint256 deadline, bool
bypassZapping ) public nonReentrant returns (uint256 addedAmountA, uint256
addedAmountB, uint256 addedLiquidity)
        {
        require( exchangeConfig.walletHasAccess(msg.sender), "Sender does not have
exchange access" );

        // Remember the initial underlying token balances of this contract so we can
later determine if any of the user's tokens are later unused for adding liquidity
and should be sent back.
        uint256 startingBalanceA = tokenA.balanceOf( address(this) );
        uint256 startingBalanceB = tokenB.balanceOf( address(this) );

        // Transfer the specified maximum amount of tokens from the user
        tokenA.safeTransferFrom(msg.sender, address(this), maxAmountA );
        tokenB.safeTransferFrom(msg.sender, address(this), maxAmountB );

        // Zap in the specified liquidity (passing the specified bypassZapping as
well).
        // The added liquidity will be owned by this contract. (external call)
        tokenA.approve( address(pools), maxAmountA );
```

```
        tokenB.approve( address(pools), maxAmountB );
        (addedAmountA, addedAmountB, addedLiquidity) = pools.dualZapInLiquidity(
tokenA, tokenB, maxAmountA, maxAmountB, minLiquidityReceived, deadline,
bypassZapping );

        // Avoid stack too deep
                {
                (bytes32 poolID,) = PoolUtils._poolID( tokenA, tokenB );

                // Increase the user's liquidity share by the amount of addedLiquidity.
                // Cooldown is specified to prevent reward hunting (ie - quickly
depositing and withdrawing large amounts of liquidity to snipe rewards)
                // Here the pool will be confirmed as whitelisted as well.
                _increaseUserShare( msg.sender, poolID, addedLiquidity, true );
                }

        // If any of the user's tokens were not used for in the dualZapInLiquidity,
then send them back
        uint256 unusedTokensA = tokenA.balanceOf( address(this) ) - startingBalanceA;
        if ( unusedTokensA > 0 )
                tokenA.safeTransfer( msg.sender, unusedTokensA );

        uint256 unusedTokensB = tokenB.balanceOf( address(this) ) - startingBalanceB;
        if ( unusedTokensB > 0 )
                tokenB.safeTransfer( msg.sender, unusedTokensB );
        }
```

Figure 13.1: src/staking/Liquidity.sol#L36-L72

Exploit Scenario

Alice attempts to add liquidity, but due to improper approval amounts, her transaction is reverted.

Recommendations

Short term, update the code to approve the proper amounts.

Long term, consider adding test assertions that do not assume maximum allowance.

A. Vulnerability Categories

The following tables describe the vulnerability categories, severity levels, and difficulty levels used in this document.

Vulnerability Categories	
Category	**Description**
Access Controls	Insufficient authorization or assessment of rights
Auditing and Logging	Insufficient auditing of actions or logging of problems
Authentication	Improper identification of users
Configuration	Misconfigured servers, devices, or software components
Cryptography	A breach of system confidentiality or integrity
Data Exposure	Exposure of sensitive information
Data Validation	Improper reliance on the structure or values of data
Denial of Service	A system failure with an availability impact
Error Reporting	Insecure or insufficient reporting of error conditions
Patching	Use of an outdated software package or library
Testing	Insufficient test methodology or test coverage
Timing	Race conditions or other order-of-operations flaws
Undefined Behavior	Undefined behavior triggered within the system

Severity Levels	
Severity	**Description**
Informational	The issue does not pose an immediate risk but is relevant to security best practices.
Undetermined	The extent of the risk was not determined during this engagement.
Low	The risk is small or is not one the client has indicated is important.
Medium	User information is at risk; exploitation could pose reputational, legal, or moderate financial risks.
High	The flaw could affect numerous users and have serious reputational, legal, or financial implications.

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploitation was not determined during this engagement.
Low	The flaw is well known; public tools for its exploitation exist or can be scripted.
Medium	An attacker must write an exploit or will need in-depth knowledge of the system.
High	An attacker must have privileged access to the system, may need to know complex technical details, or must discover other weaknesses to exploit this issue.

B. Code Maturity Categories

The following tables describe the code maturity categories and rating criteria used in this document.

Code Maturity Categories	
Category	**Description**
Arithmetic	The proper use of mathematical operations and semantics
Auditing	The use of event auditing and logging to support monitoring
Authentication / Access Controls	The use of robust access controls to handle identification and authorization and to ensure safe interactions with the system
Complexity Management	The presence of clear structures designed to manage system complexity, including the separation of system logic into clearly defined functions
Cryptography and Key Management	The safe use of cryptographic primitives and functions, along with the presence of robust mechanisms for key generation and distribution
Decentralization	The presence of a decentralized governance structure for mitigating insider threats and managing risks posed by contract upgrades
Documentation	The presence of comprehensive and readable codebase documentation
Transaction Ordering	The system's resistance to transaction-ordering attacks
Low-Level Manipulation	The justified use of inline assembly and low-level calls
Testing and Verification	The presence of robust testing procedures (e.g., unit tests, integration tests, and verification methods) and sufficient test coverage

Rating Criteria	
Rating	**Description**
Strong	No issues were found, and the system exceeds industry standards.
Satisfactory	Minor issues were found, but the system is compliant with best practices.
Moderate	Some issues that may affect system safety were found.
Weak	Many issues that affect system safety were found.
Missing	A required component is missing, significantly affecting system safety.
Not Applicable	The category is not applicable to this review.
Not Considered	The category was not considered in this review.
Further Investigation Required	Further investigation is required to reach a meaningful conclusion.

C. System Diagrams

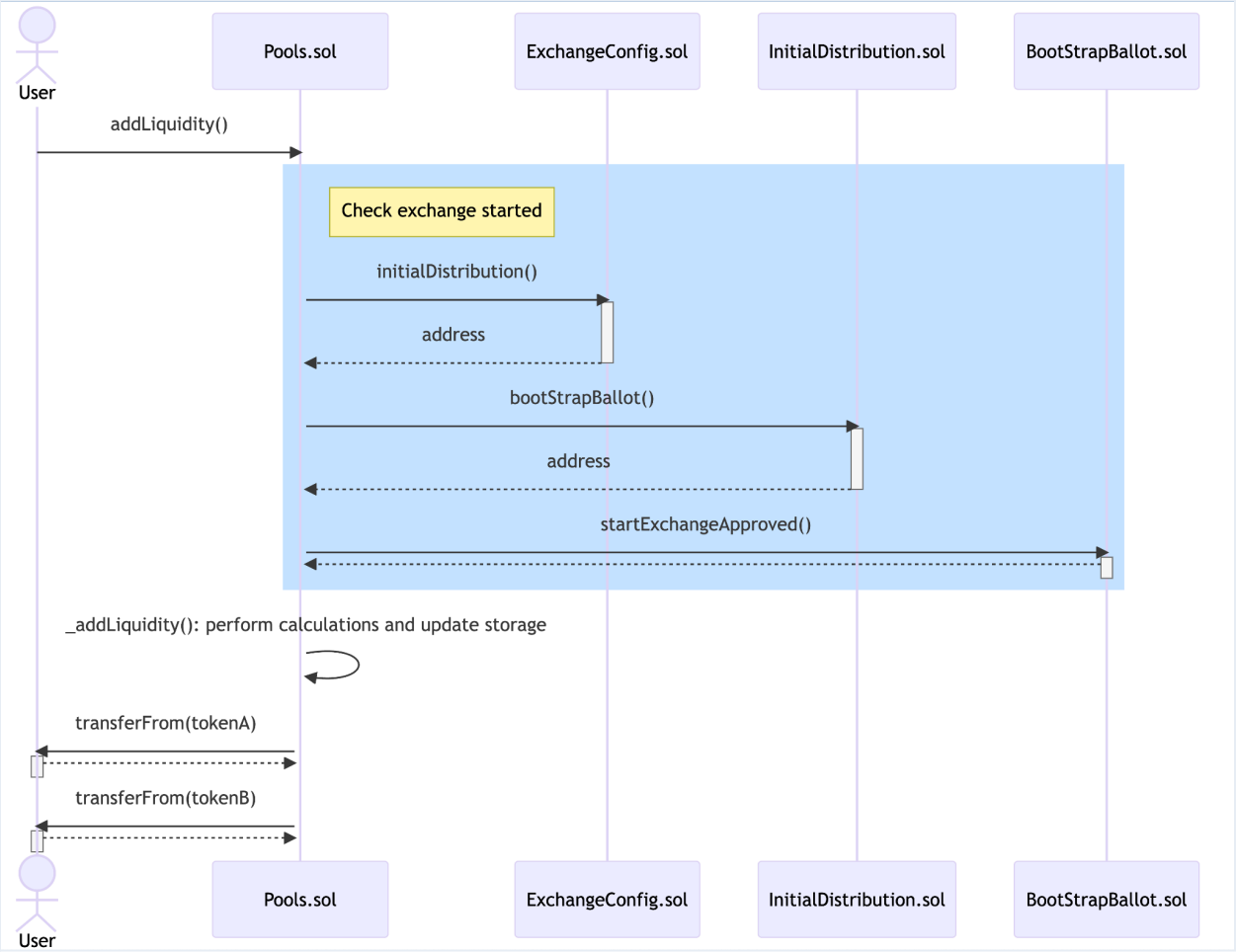

Figure C.1: A high-level system diagram of the `addLiquidity` function's logic

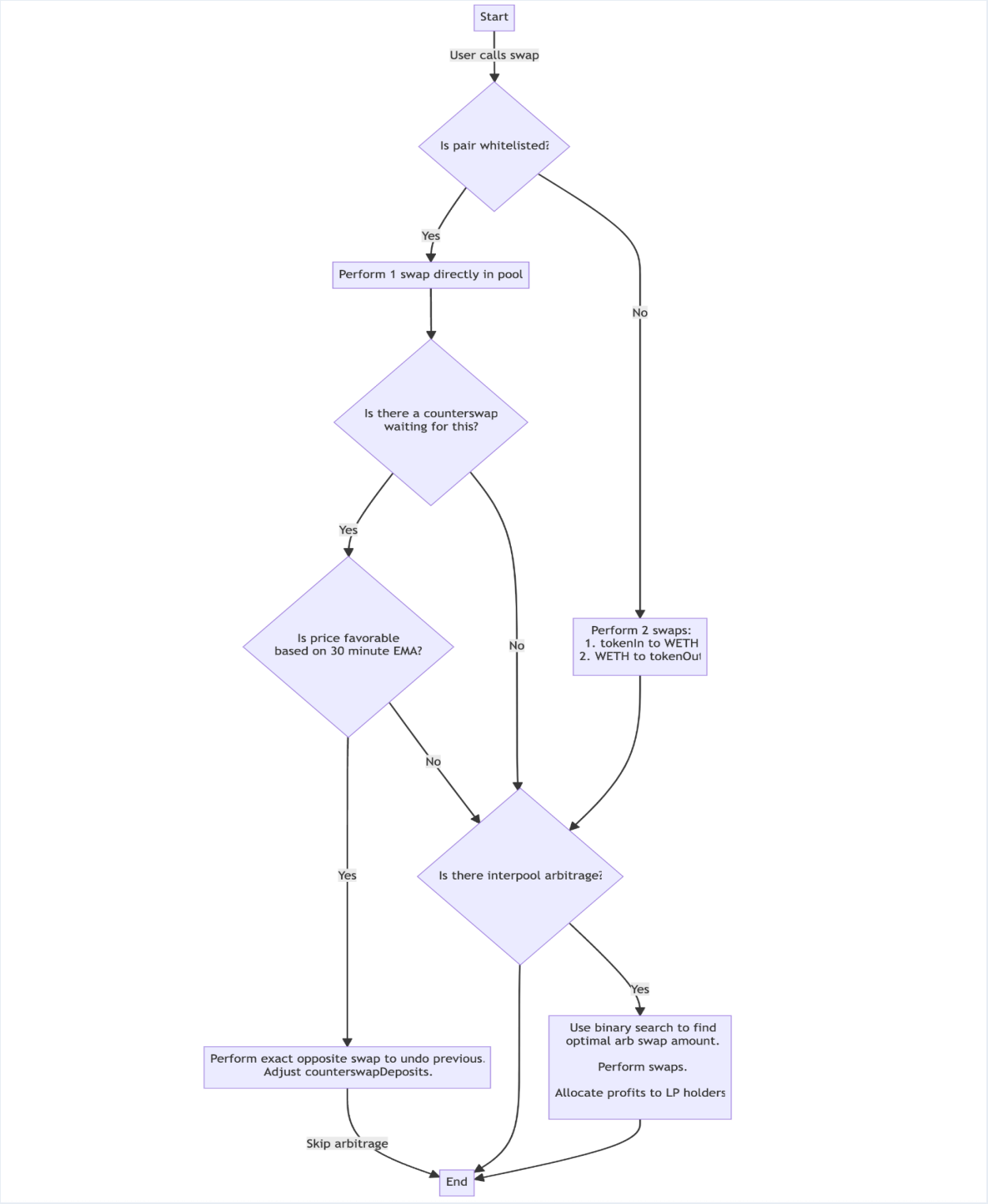

Figure C.2: A high-level system diagram of the swap logic

D. Token Integration Checklist

The following checklist provides recommendations for interactions with arbitrary tokens. Every unchecked item should be justified and its associated risks understood. For an up-to-date version of the checklist, see `crytic/building-secure-contracts`.

For convenience, all Slither utilities can be run directly on a token address, such as the following:

```
slither-check-erc 0xdac17f958d2ee523a2206206994597c13d831ec7 TetherToken --erc erc20
slither-check-erc 0x06012c8cf97BEaD5deAe237070F9587f8E7A266d KittyCore --erc erc721
```

To follow this checklist, use the following output from Slither for the token:

```
slither-check-erc [target] [contractName] [optional: --erc ERC_NUMBER]
slither [target] --print human-summary
slither [target] --print contract-summary
slither-prop . --contract ContractName # requires configuration, and use of Echidna
and Manticore
```

General Considerations

- ❏ **The contract has a security review.** Avoid interacting with contracts that lack a security review. Check the length of the assessment (i.e., the level of effort), the reputation of the security firm, and the number and severity of the findings.
- ❏ **You have contacted the developers.** You may need to alert their team to an incident. Look for appropriate contacts on `blockchain-security-contacts`.
- ❏ **They have a security mailing list for critical announcements.** Their team should advise users when critical issues are found or when upgrades occur.

Contract Composition

- ❏ **The contract avoids unnecessary complexity.** The token should be a simple contract; a token with complex code requires a higher standard of review. Use Slither's `human-summary` printer to identify complex code.
- ❏ **The contract uses `SafeMath` or Solidity 0.8.0+.** Contracts that do not use `SafeMath` require a higher standard of review. Inspect the contract by hand for `SafeMath`/Solidity 0.8.0+ usage.
- ❏ **The contract has only a few non-token-related functions.** Non-token-related functions increase the likelihood of an issue in the contract. Use Slither's `contract-summary` printer to broadly review the code used in the contract.

- ❏ **The token has only one address.** Tokens with multiple entry points for balance updates can break internal bookkeeping based on the address (e.g., `balances[token_address][msg.sender]` may not reflect the actual balance).

Owner Privileges

- ❏ **The token is not upgradeable.** Upgradeable contracts may change their rules over time. Use Slither's `human-summary` printer to determine whether the contract is upgradeable.
- ❏ **The owner has limited minting capabilities.** Malicious or compromised owners can misuse minting capabilities. Use Slither's `human-summary` printer to review minting capabilities, and consider manually reviewing the code.
- ❏ **The token is not pausable.** Malicious or compromised owners can trap contracts relying on pausable tokens. Identify pausable code by hand.
- ❏ **The owner cannot denylist the contract.** Malicious or compromised owners can trap contracts relying on tokens with a denylist. Identify denylisting features by hand.
- ❏ **The team behind the token is known and can be held responsible for misuse.** Contracts with anonymous development teams or teams that reside in legal shelters require a higher standard of review.

ERC-20 Tokens

ERC-20 Conformity Checks

Slither includes a utility, `slither-check-erc`, that reviews the conformance of a token to many related ERC standards. Use `slither-check-erc` to review the following:

- ❏ **`Transfer` and `transferFrom` return a Boolean.** Several tokens do not return a Boolean on these functions. As a result, their calls in the contract might fail.
- ❏ **The `name`, `decimals`, and `symbol` functions are present if used.** These functions are optional in the ERC-20 standard and may not be present.
- ❏ **`Decimals` returns a `uint8`.** Several tokens incorrectly return a `uint256`. In such cases, ensure that the value returned is less than 255.
- ❏ **The token mitigates the known ERC-20 race condition.** The ERC-20 standard has a known ERC-20 race condition that must be mitigated to prevent attackers from stealing tokens.

Slither includes a utility, `slither-prop`, that generates unit tests and security properties that can discover many common ERC flaws. Use `slither-prop` to review the following:

- ❏ **The contract passes all unit tests and security properties from `slither-prop`.** Run the generated unit tests and then check the properties with Echidna and Manticore.

Risks of ERC-20 Extensions

The behavior of certain contracts may differ from the original ERC specification. Conduct a manual review of the following conditions:

- ❏ **The token is not an ERC-777 token and has no external function call in `transfer` or `transferFrom`.** External calls in the transfer functions can lead to reentrancies.
- ❏ **`Transfer` and `transferFrom` should not take a fee.** Deflationary tokens can lead to unexpected behavior.
- ❏ **Potential interest earned from the token is accounted for.** Some tokens distribute interest to token holders. This interest may be trapped in the contract if not accounted for.

Token Scarcity

Reviews of token scarcity issues must be executed manually. Check for the following conditions:

- ❏ **The supply is owned by more than a few users.** If a few users own most of the tokens, they can influence operations based on the tokens' repartition.
- ❏ **The total supply is sufficient.** Tokens with a low total supply can be easily manipulated.
- ❏ **The tokens are in more than a few exchanges.** If all the tokens are in one exchange, a compromise of the exchange could compromise the contract relying on the token.
- ❏ **Users understand the risks associated with a large amount of funds or flash loans.** Contracts relying on the token balance must account for attackers with a large amount of funds or attacks executed through flash loans.
- ❏ **The token does not allow flash minting.** Flash minting can lead to substantial swings in the balance and the total supply, which necessitate strict and comprehensive overflow checks in the operation of the token.

E. Fix Review Results

When undertaking a fix review, Trail of Bits reviews the fixes implemented for issues identified in the original report. This work involves a review of specific areas of the source code and system configuration, not comprehensive analysis of the system.

From December 4 to December 6, 2023, Trail of Bits reviewed the fixes and mitigations implemented by Salty.IO for the issues identified in this report. We reviewed each fix to determine its effectiveness in resolving the associated issue.

Our review process was impacted by the format of the fixes provided. Normally, we recommend one PR per fix. For this review, the fixes were included along with 147 commits updating over 15,000 lines of code, most of which were not related to the fix review. The client provided a response document containing snippets of updated code along with explanatory comments (appendix G). Our review of the fixes was focused on the current state of the code (as of commit `9cb834f`) and was guided by this response document. In the Detailed Fix Review Results section below, the commits listed for each finding may contain additional changes unrelated to any specific issue remediation. *These additional changes were not reviewed by us.*

Due to the number of changes observed, we suggest conducting an additional security review of the entire protocol before going to production.

Salty has resolved 12 of the 13 issues described in this report. Detailed information about the unresolved issue can be found below in the Detailed Fix Review Results section.

ID	Title	Status
1	Risk of denial-of-service attacks on token whitelisting process	**Resolved**
2	Insufficient event generation	**Resolved**
3	Transactions to add liquidity may be front run	**Resolved**
4	Whitelisted pools may exceed the maximum allowed	**Unresolved**

5	Any user can add liquidity to any pool and bypass the token whitelist	**Resolved**
6	Liquidation fee is volatile and may be manipulated	**Resolved**
7	Collateral contract deployment results in permanent loss of rewards	**Resolved**
8	Collateral can be withdrawn without repaying USDS loan	**Resolved**
9	Lack of chain ID validation allows signature reuse across forks	**Resolved**
10	Chainlink oracles could return stale price data	**Resolved**
11	Lack of timely price feed updates may result in loss of funds	**Resolved**
12	USDS stablecoin may become undercollateralized	**Resolved**
13	Zap operations may approve an incorrect number of tokens, leading to reversion	**Resolved**

Detailed Fix Review Results

TOB-SALTY-1: Risk of denial-of-service attacks on token whitelisting process
Resolved in commits `689f7de`, `d3621a9`, and `ca1a859`. The client provided the following additional context:

> *To ensure that some adequate total stake exists to be used as a reference, proposals are delayed for the first 45 days after deployment.*
>
> *The requirement for making proposals has also changed from a proposal cost to a minimum amount of staked SALT.*
>
> *Additionally, users are only able to have one active proposal at a time, which, combined with xSALT's default one year unstaking period, should help to limit spamming of the proposals.*

We also recommend documenting these changes to inform future development.

TOB-SALTY-2: Insufficient event generation
Resolved in commits `cadd4b` and `e38252f`. The client provided the following additional context:

> *Comprehensive events are now included throughout the contracts.*

TOB-SALTY-3: Transactions to add liquidity may be front run
Resolved in commit `7346286`. The client provided the following additional context:

> *`Pools.sol` now checks to make sure that the removal of liquidity does not drive reserves below dust. Additionally, checks are made following swaps to ensure that reserves remain above dust.*

TOB-SALTY-4: Whitelisted pools may exceed the maximum allowed
Unresolved. The client acknowledged the risk and provided the following additional context:

> *Yes, the number of whitelisted pools could stay above `maximumWhitelistedPools` if `maximumWhitelistedPools` is decreased below the current number of whitelisted pools. The new limit would then prevent the whitelisting of additional pools until some existing pools are unwhitelisted.*

We also recommend documenting this issue to inform future developers and governance participants.

TOB-SALTY-5: Any user can add liquidity to any pool and bypass the token whitelist
Resolved in commit `7fd4ed4`. The client provided the following additional context:

> *`Pools.addLiquidity` and `Pools.removeLiquidity` are now only callable from the `CollateralAndLiquidity` contract itself.*

TOB-SALTY-6: Liquidation fee is volatile and may be manipulated
Resolved in commit `012ce88`. The client provided the following additional context:

> *The liquidator now receives 5% of both the liquidated WBTC and 5% of the liquidated WETH.*

TOB-SALTY-7: Collateral contract deployment results in permanent loss of rewards
Resolved in commit `7fd4ed4`. The client provided the following additional context:

> *`Liquidity.sol` and `Collateral.sol` are no longer deployed as separate contracts and are now deployed as only `CollateralAndLiquidity.sol`. As such, the `liquidityRewardsEmitter` now has full access to all liquidity pools in `CollateralAndLiquidity.sol`, including WBTC/WETH.*

TOB-SALTY-8: Collateral can be withdrawn without repaying USDS loan
Resolved in commit `52ef63c`. The client provided the following additional context:

> *`_withdrawLiquidityAndClaim` now contains the withdraw functionality, with `Liquidity.withdrawLiquidityAndClaim` disallowing withdrawal from the WBTC/WETH collateral pool.*

TOB-SALTY-9: Lack of chain ID validation allows signature reuse across forks
Resolved in commit `fe995a2`. The client provided the following additional context:

> *The chain ID is now included in the `messageHash` for signature verification.*

TOB-SALTY-10: Chainlink oracles could return stale price data
Resolved in commit `4a10672`. The client provided the following additional context:

> *Chainlink prices are now excluded if they are older than the recommended 60 minute heartbeat.*

TOB-SALTY-11: Lack of timely price feed updates may result in loss of funds
Resolved in commit `1174658`. The client provided the following additional context:

> *The `PriceAggregator` no longer caches prices and now returns the current price on each call.*

TOB-SALTY-12: USDS stablecoin may become undercollateralized
Resolved in commit 6a1dc7a. The client provided the following additional context:

> *Counterswap has been replaced with direct swaps to facilitate faster liquidation of WBTC and WETH. Simulation has shown that the atomic arbitrage mechanic limits the effect of sandwich attacks and allows for the direct swaps instead of the more cumbersome counterswaps.*
>
> *Additionally, a newly added `Liquidizer.sol` contract now allows withdrawal of USDS/DAI and SALT/USDS Protocol Owned Liquidity, which can then be converted to burnable USDS as an additional safety net.*

TOB-SALTY-13: Zap operations may approve an incorrect number of tokens, leading to reversion
Resolved in commit 1174658. The client provided the following additional context:

> *Approvals now happen after the zap has altered the token amounts.*

F. Fix Review Status Categories

The following table describes the statuses used to indicate whether an issue has been sufficiently addressed.

Fix Status	
Status	**Description**
Undetermined	The status of the issue was not determined during this engagement.
Unresolved	The issue persists and has not been resolved.
Partially Resolved	The issue persists but has been partially resolved.
Resolved	The issue has been sufficiently resolved.

G. Client Responses for Fix Review

These are the client responses for the fix review, which the review was based on.

1. Risk of denial-of-service attacks on token whitelisting process
The requirement for making proposals has changed from a proposal cost to a minimum amount of staked SALT. Additionally, users are only able to have one active proposal at a time, which, combined with xSALT's default one year unstaking period, should help to limit spamming of the proposals. To ensure that some adequate total stake exists to be used as a reference, proposals are delayed for the first 45 days after deployment.

From `DAOConfig.sol`:

```
// The percent of staked SALT that a user has to have to make a proposal
// Range: 0.10% to 2% with an adjustment of 0.10%
uint256 public requiredProposalPercentStakeTimes1000 = 5; // Initially 0.50%
```

From `Proposals.sol`:

```
function _possiblyCreateProposal( string memory ballotName, BallotType
ballotType, address address1, uint256 number1, string memory string1, string
memory string2 ) internal returns (uint256 ballotID)
	{
	require( block.timestamp >= firstPossibleProposalTimestamp, "Cannot
propose ballots within the first 45 days of deployment" );
	// The DAO can create confirmation proposals which won't have the below
requirements
	if ( msg.sender != address(exchangeConfig.dao() ) )
		{
		// Make sure that the sender has the minimum amount of xSALT
required to make the proposal
		uint256 totalStaked = staking.totalShares(PoolUtils.STAKED_SALT);
		require( totalStaked > 0, "Total staked cannot be zero" );
		uint256 requiredXSalt = ( totalStaked *
daoConfig.requiredProposalPercentStakeTimes1000() ) / ( 100 * 1000 );
		uint256 userXSalt = staking.userShareForPool( msg.sender,
PoolUtils.STAKED_SALT );
		require( userXSalt >= requiredXSalt, "Sender does not have enough
xSALT to make the proposal" );
		// Make sure that the user doesn't already have an active proposal
		require( ! _userHasActiveProposal[msg.sender], "Users can only
have one active proposal at a time" );
		}
```

2. Insufficient event generation
Comprehensive events are now included throughout the contracts.

3. Transactions to add liquidity may be front run
`Pools.sol` now checks to make sure that the removal of liquidity does not drive reserves below dust. Additionally, checks are made following swaps to ensure that reserves remain above dust.

From `Pools.sol::removeLiquidity`:

```
// Make sure that removing liquidity doesn't drive the reserves below DUST
if ( reserves.reserve0 < (reclaimedA + PoolUtils.DUST) )
       reclaimedA = reserves.reserve0 - PoolUtils.DUST;
if ( reserves.reserve1 < (reclaimedB + PoolUtils.DUST) )
       reclaimedB = reserves.reserve1 - PoolUtils.DUST;
reserves.reserve0 -= uint128(reclaimedA);
reserves.reserve1 -= uint128(reclaimedB);
```

From `Pools.sol::_adjustReservesForSwap`:

```
// Make sure that the reserves after swap are above DUST
require( (reserve0 >= PoolUtils.DUST) && (reserve1 >= PoolUtils.DUST),
"Insufficient reserves after swap");
```

4. Whitelisted pools may exceed the maximum allowed

Yes, the number of whitelisted pools could stay above `maximumWhitelistedPools` if `maximumWhitelistedPools` is decreased below the current number of whitelisted pools. The new limit would then prevent the whitelisting of additional pools until some existing pools are unwhitelisted.

5. Any user can add liquidity to any pool and bypass the token whitelist

`Liquidity.sol` and `Collateral.sol` are no longer deployed as separate contracts and are now deployed as only `CollateralAndLiquidity.sol`.

`Pools.addLiquidity` and `Pools.removeLiquidity` are now only callable from the `CollateralAndLiquidity` contract itself.

From `Pools.sol::addLiquidity`:

```
require( msg.sender == address(collateralAndLiquidity), "Pools.addLiquidity is
only callable from the CollateralAndLiquidity contract" );
```

From `Pools.sol::removeLiquidity`:

```
require( msg.sender == address(collateralAndLiquidity), "Pools.removeLiquidity
is only callable from the CollateralAndLiquidity contract" );
```

6. Liquidation fee is volatile and may be manipulated

The liquidator now receives 5% of both the liquidated WBTC and 5% of the liquidated WETH.

From `CollateralAndLiquidity::liquidateUser`:

```
uint256 rewardedWBTC = (reclaimedWBTC * rewardPercent) / 100;
uint256 rewardedWETH = (reclaimedWETH * rewardPercent) / 100;
// Make sure the value of the rewardAmount is not excessive
uint256 rewardValue = underlyingTokenValueInUSD( rewardedWBTC, rewardedWETH );
// in 18 decimals
uint256 maxRewardValue = stableConfig.maxRewardValueForCallingLiquidation(); //
18 decimals
if ( rewardValue > maxRewardValue )
       {
```

```
        rewardedWBTC = (rewardedWBTC * maxRewardValue) / rewardValue;
        rewardedWETH = (rewardedWETH * maxRewardValue) / rewardValue;
        }
// Reward the caller
wbtc.safeTransfer( msg.sender, rewardedWBTC );
weth.safeTransfer( msg.sender, rewardedWETH );
```

7. Collateral contract deployment results in permanent loss of rewards
`Liquidity.sol` and `Collateral.sol` are no longer deployed as separate contracts and are now deployed as only `CollateralAndLiquidity.sol`.

As such, the `liquidityRewardsEmitter` now has full access to all liquidity pools in `CollateralAndLiquidity.sol`, including WBTC/WETH.

8. Collateral can be withdrawn without repaying USDS loan
`_withdrawLiquidityAndClaim` now contains the withdraw functionality, with `Liquidity.withdrawLiquidityAndClaim` disallowing withdrawal from the WBTC/WETH collateral pool.

From `Liquidity.sol`:
```
// Public wrapper for withdrawing liquidity which prevents the direct withdrawal
from the collateral pool.
// CollateralAndLiquidity.withdrawCollateralAndClaim bypasses this and calls
_withdrawLiquidityAndClaim directly.
// No exchange access required for withdrawals.
   function withdrawLiquidityAndClaim( IERC20 tokenA, IERC20 tokenB, uint256
liquidityToWithdraw, uint256 minReclaimedA, uint256 minReclaimedB, uint256
deadline ) external nonReentrant ensureNotExpired(deadline) returns (uint256
reclaimedA, uint256 reclaimedB)
        {
        require( PoolUtils._poolIDOnly( tokenA, tokenB ) != collateralPoolID,
"Stablecoin collateral cannot be withdrawn via
Liquidity.withdrawLiquidityAndClaim" );
        return _withdrawLiquidityAndClaim(tokenA, tokenB, liquidityToWithdraw,
minReclaimedA, minReclaimedB);
        }
```

9. Lack of chain ID validation allows signature reuse across forks
The chain ID is now included in the `messageHash` for signature verification.

From `AccessManager.sol::_verifyAccess`:
```
bytes32 messageHash = keccak256(abi.encodePacked(block.chainid, geoVersion,
wallet));
```

From `BootstrapBallot.sol::vote`:
```
bytes32 messageHash = keccak256(abi.encodePacked(block.chainid, msg.sender));
```

10. Chainlink oracles could return stale price data

Chainlink prices are now excluded if they are older than the recommended 60 minute heartbeat.

From `CoreChainlinkFeed.sol::latestChainlinkPrice`:

```
// Make sure that the Chainlink price update has occurred within its 60 minute
heartbeat
// https://docs.chain.link/data-feeds#check-the-timestamp-of-the-latest-answer
uint256 answerDelay = block.timestamp - _answerTimestamp;
if ( answerDelay > 60 minutes )
       price = 0;
```

11. Lack of timely price feed updates may result in loss of funds

The `PriceAggregator` no longer caches prices and now returns the current price on each call.

From `PriceAggregator`:

```
// Return the current BTC price (with 18 decimals)
function getPriceBTC() external view returns (uint256 price)
       {
       uint256 price1 = _getPriceBTC(priceFeed1);
       uint256 price2 = _getPriceBTC(priceFeed2);
       uint256 price3 = _getPriceBTC(priceFeed3);
       price = _aggregatePrices(price1, price2, price3);
       require (price != 0, "Invalid BTC price" );
       }
// Return the current ETH price (with 18 decimals)
function getPriceETH() external view returns (uint256 price)
       {
       uint256 price1 = _getPriceETH(priceFeed1);
       uint256 price2 = _getPriceETH(priceFeed2);
       uint256 price3 = _getPriceETH(priceFeed3);
       price = _aggregatePrices(price1, price2, price3);
       require (price != 0, "Invalid ETH price" );
       }
```

12. USDS stablecoin may become undercollateralized

Counterswap has been replaced with direct swaps to facilitate faster liquidation of WBTC and WETH. Simulation has shown that the atomic arbitrage mechanic limits the effect of sandwich attacks and allows for the direct swaps instead of the more cumbersome counterswaps.

Additionally, a newly added `Liquidizer.sol` contract now allows withdrawal of USDS/DAI and SALT/USDS Protocol Owned Liquidity, which can then be converted to burnable USDS as an additional safety net.

From `PoolUtils.sol`:

```
// Swaps tokens internally within the protocol with amountIn limited to be a
certain percent of the reserves.
// The limit, combined with atomic arbitrage makes sandwich attacks on this swap
less profitable (even with no slippage being specified).
```

```
// This is due to the first swap of the sandwich attack being offset by atomic
arbitrage within its same transaction.
// This effectively reverses some of the initial swap of the attack and creates
an initial loss for the attacker proportional to the size of that swap (if they
were to swap back immediately).
// Simulations (see Sandwich.t.sol) show that when sandwich attacks are used, the
arbitrage earned by the protocol sometimes exceeds any amount lost due to the
sandwich attack itself.
// The largest swap loss seen in the simulations was 1.8% (under an unlikely
scenario).   More typical losses would be 0-1%.
// The actual swap loss (taking arbitrage profits generated by the sandwich swaps
into account) is dependent on the multiple pool reserves involved in the
arbitrage (which are encouraged by rewards distribution to create more reasonable
arbitrage opportunities).
// Also, the protocol awards a default 5% of pending arbitrage profits to users
that call Upkeep.performUpkeep().
// If sandwiching performUpkeep (where these internal swaps happen) is profitable
it would encourage "attackers" to call performUpkeep more often.
// With that in mind, the DAO could choose to lower the default 5% reward for
performUpkeep callers - effectively making sandwich "attacks" part of the
performUpkeep mechanic itself.
function _placeInternalSwap( IPools pools, IERC20 tokenIn, IERC20 tokenOut,
uint256 amountIn, uint256 maximumInternalSwapPercentTimes1000 ) internal
returns (uint256 swapAmountIn, uint256 swapAmountOut)
	{
	if ( amountIn == 0 )
		return (0, 0);
	(uint256 reservesIn,) = pools.getPoolReserves( tokenIn, tokenOut );
	uint256 maxAmountIn = reservesIn * maximumInternalSwapPercentTimes1000 /
(100000);
	if ( amountIn > maxAmountIn )
		amountIn = maxAmountIn;
	swapAmountIn = amountIn;
	swapAmountOut = pools.depositSwapWithdraw(tokenIn, tokenOut, amountIn, 0,
block.timestamp );
	}
```

From `Liquidizer::_possiblyBurnUSDS`:

```
// As there is a shortfall in the amount of USDS that can be burned, liquidate
some Protocol Owned Liquidity and
// send the underlying tokens here to be swapped to USDS
dao.withdrawPOL(salt, usds, PERCENT_POL_TO_WITHDRAW);
dao.withdrawPOL(dai, usds, PERCENT_POL_TO_WITHDRAW);
```

From `DAO.sol`:

```
// Withdraws the specified amount of the Protocol Owned Liquidity from the DAO
and sends the underlying tokens to the Liquidizer to be burned as USDS as needed.
// Called when the amount of recovered USDS from liquidating a user's WBTC/WETH
collateral is insufficient to cover burning the USDS that they had borrowed.
// Only callable from the Liquidizer contract.
function withdrawPOL( IERC20 tokenA, IERC20 tokenB, uint256 percentToLiquidate
) external
	{
```

```
        require(msg.sender == address(liquidizer),
"DAO.withdrawProtocolOwnedLiquidity is only callable from the Liquidizer
contract" );
        bytes32 poolID = PoolUtils._poolIDOnly(tokenA, tokenB);
        uint256 liquidityHeld = collateralAndLiquidity.userShareForPool(
address(this), poolID );
        if ( liquidityHeld == 0 )
                return;
        uint256 liquidityToWithdraw = (liquidityHeld * percentToLiquidate) / 100;
        // Withdraw the specified Protocol Owned Liquidity
        (uint256 reclaimedA, uint256 reclaimedB) =
collateralAndLiquidity.withdrawLiquidityAndClaim(tokenA, tokenB,
liquidityToWithdraw, 0, 0, block.timestamp );
        // Send the withdrawn tokens to the Liquidizer so that the tokens can be
swapped to USDS and burned as needed.
        tokenA.safeTransfer( address(liquidizer), reclaimedA );
        tokenB.safeTransfer( address(liquidizer), reclaimedB );
        emit POLWithdrawn(address(tokenA), address(tokenB), reclaimedA,
reclaimedB);
        }
```

13. Zap operations may approve an incorrect amount of tokens, leading to reversion

Approvals now happen after the zap has altered the token amounts.

From `Liquidity.sol::_depositLiquidityAndIncreaseShare`:

```
// Balance the token amounts by swapping one to the other before adding the
liquidity?
if ( useZapping )
        (maxAmountA, maxAmountB) = _dualZapInLiquidity(tokenA, tokenB,
maxAmountA, maxAmountB );
// Approve the liquidity to add
tokenA.approve( address(pools), maxAmountA );
tokenB.approve( address(pools), maxAmountB );
```