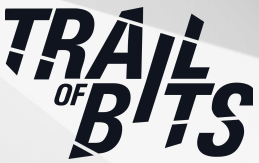

Acronym Foundation

Security Assessment

December 13, 2023

Prepared for:
Acronym Foundation Team
Acronym Foundation

Prepared by: **Priyanka Bose and Anish Naik**

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

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

Project Summary

Contact Information

The following managers were associated with this project:

Dan Guido, Account Manager
dan@trailofbits.com

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

The following engineers were associated with this project:

Priyanka Bose, Consultant
priyanka.bose@trailofbits.com

Anish Naik, Consultant
anish.naik@trailofbits.com

Project Timeline

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

Date	Event
September 8, 2023	Pre-project kickoff call
September 15, 2023	Status update meeting #1
September 22, 2023	Delivery of report draft
September 25, 2023	Report readout meeting
October 12, 2023	Delivery of comprehensive report
December 13, 2023	Delivery of comprehensive report with fix review

Executive Summary

Engagement Overview

Acronym Foundation engaged Trail of Bits to review the security of the Anvil protocol which is a lending and borrowing platform.

A team of two consultants conducted the review from September 11 to September 22, 2023, for a total of four engineer-weeks of effort. Our testing efforts focused on identifying any instances that could result in a loss of funds, unexpected liquidations, and arithmetic inconsistencies. With full access to source code and documentation, we performed static and dynamic testing of the codebase, using automated and manual processes.

Observations and Impact

The Anvil protocol is a lending and borrowing platform that facilitates the transparent use of collateral by approved parties within the system. Creditors will open a line of credit that is backed by collateral for a specified beneficiary to use before an expiration time. We identified two core patterns that led to the issues identified in this report.

First, the logic surrounding partial redemptions led to issues TOB-ANVIL-1, TOB-ANVIL-4, and TOB-ANVIL-5. These issues allow the theft of funds and unexpected liquidations, which pose a significant risk to the protocol's operations. These problems suggest a systemic pattern involving the logic surrounding partial redemptions, and there may be additional related issues that remain undetected at present. Second, TOB-ANVIL-2 and TOB-ANVIL-6 highlight weaknesses in the unit testing suite. Increasing branch coverage, testing all edge cases, and adding more thorough post-condition checks would significantly improve the testing suite.

Despite these issues, the documentation and inline comments highlight the ANVIL team's strong understanding of their system's core invariants and how each protocol operation affects the state of the system. Integrating fuzz testing to dynamically test these invariants would also significantly improve the protocol's security posture.

Recommendations

Based on the codebase maturity evaluation and findings identified during the security review, Trail of Bits recommends that Acronym Foundation 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.
- **Rework the logic surrounding partial redemption.** Due to several issues connected to partial redemptions, conduct a thorough review and overhaul of the

associated logic. This will help address the identified issues and enhance the functionality and reliability of the partial redemption process.

- **Add complex unit and fuzzing tests into the codebase.** Add robust unit tests that encompass all the arithmetic and rounding computations within the system. Ensure that the unit tests have thorough post-condition checks to evaluate all state changes. Additionally, integrate a dynamic fuzzing solution like Echidna to test critical function- and system-level invariants.

Finding Severities and Categories

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

EXPOSURE ANALYSIS

Severity	*Count*
High	4
Medium	1
Low	1
Informational	1
Undetermined	0

CATEGORY BREAKDOWN

Category	*Count*
Data Validation	7

Project Goals

The engagement was scoped to provide a security assessment of the Acronym Foundation's DeFi protocol. Specifically, we sought to answer the following non-exhaustive list of questions:

- Is it possible to drain funds from the `Collateral` vault and `LetterOfCredit` contract?
- Are there any arithmetic rounding issues affecting the code?
- Are there appropriate access controls on critical functions?
- Are there any reentrancy or denial-of-service attack vectors?
- Are the inputs validated correctly?
- Can letters of credit (LOCs) be forced to become insolvent?
- Does the system properly calculate fees?
- Is the pricing for token pairs computed correctly?
- Is it possible to front-run transactions and negatively affect the system?

Project Targets

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

DeFi protocol

Repository	https://github.com/AmperaFoundation/sol-contracts
Version	`94d6dc5a3adfddbbd7229c8e087d381c95387dd8`
Type	Solidity
Platform	EVM

*The repository above has since been renamed
https://github.com/AcronymFoundation/anvil-contracts

Project Coverage

This section provides an overview of the analysis coverage of the review, as determined by our high-level engagement goals. Our approaches and results included the following:

- **`Collateral` vault contract.** This contract allows users to deposit or withdraw funds that can later be used as collateral by a collateralizable contract such as `LetterOfCredit`. The users can directly interact with this contract to check balances, reserve collateral, and approve one or more collateralizable contracts to access their collateral. We verified the deposit and withdrawal workflows thoroughly to make sure that no funds are lost from this contract. Additionally, we checked whether all necessary state variable updates are executed accurately across all potential workflows. We also thoroughly examined all critical functions within the contract to identify any potential access control violations.

- **`LetterOfCredit` contract.** This contract primarily serves the purpose of enabling a user to generate an LOC that can be redeemed by a beneficiary before a specified expiration date set by the creator. To create the LOC, the user reserves collateral based on a predetermined collateralization factor and has the option to convert the collateral asset into the credited asset, which the beneficiary can then receive. If the LOC becomes unhealthy, it allows any user to liquidate the collateral. We performed a manual review of this contract and investigated the following:

    - We reviewed the creation of LOCs to look for any possible deviations in the implementation from the documentation. We found that LOCs created from an already expired LOC can last longer than the allowed duration (TOB-ANVIL-2).

    - We verified the redemption logic to look for any potential loss of funds from the contract. Additionally, we checked whether the critical state variables are updated to reflect the redemption amount during partial redemptions. Our analysis led to the discovery that partial redemptions allow users to drain funds from the contract (TOB-ANVIL-4 and TOB-ANVIL-5). Furthermore, we identified an issue with the correct updating of critical state variables during partial redemptions, which resulted in the inability to liquidate an unhealthy LOC (TOB-ANVIL-1).

- **`Pricing` and `PythPriceOracle` contracts.** These contracts are responsible for retrieving prices for collateral asset and credit asset token pairs. These price values are later used by the `LetterOfCredit` contract to reserve, claim, or liquidate collateral. During the review, we verified the accuracy of price calculations for token pairs and conducted a best-effort review of concerns related to rounding errors.

This led to the discovery that incorrect price reporting can allow unexpected liquidations (TOB-ANVIL-6).

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:

- While we expended a significant effort on reviewing the rounding logic surrounding the token pair price computation, this code should also be tested dynamically. Integrating a fuzzing solution, such as Echidna, to validate precision and rounding errors would significantly improve the assurance surrounding this logic.

Automated Testing

Trail of Bits uses automated techniques to extensively test the security properties of software. We use both open-source static analysis and fuzzing utilities, along with tools developed in house, to perform automated testing of source code and compiled software.

In this assessment, we used Echidna, a smart contract fuzzer that can rapidly test security properties via malicious, coverage-guided test case generation, to check various system states.

Test Results

The results of this focused testing are detailed below.

`Collateral.sol`. The `Collateral.sol` file holds the `Collateral` contract, which is used by creditors (creators) to deposit and withdraw collateral. Additionally, the `LetterOfCredit` contract uses the `Collateral` contract to reserve, claim, and release collateral for LOCs. We used Echidna to test the contract's deposit and withdrawal workflows.

Property	Tool	Result
The `depositAndApprove` function should increase the sender's available collateral balance.	Echidna	**Passed**
The `depositAndApprove` function should not change the sender's reserved balance.	Echidna	**Passed**
The `depositAndApprove` function should decrease the sender's collateral token balance by the amount that was sent.	Echidna	**Passed**
The `depositAndApprove` function should increase the `Collateral` contract's collateral token by the amount that was sent.	Echidna	**Passed**
The `depositAndApprove` function should increase the cumulative user balance for the collateral token by the amount that was sent.	Echidna	**Passed**
The `depositAndApprove` function should approve the associated `LetterOfCredit` contract to reserve, claim, and release the sender's collateral funds.	Echidna	**Passed**

The `withdraw` function should decrease the sender's available collateral balance.	Echidna	**Passed**
The `withdraw` function should not change the sender's reserved balance.	Echidna	**Passed**
The `withdraw` function should increase the sender's collateral token balance by the amount that was requested minus the protocol fee.	Echidna	**Passed**
The `withdraw` function should decrease the `Collateral` contract's collateral token by the amount that was requested minus the protocol fee.	Echidna	**Passed**
The `withdraw` function should increase the cumulative user balance for the collateral token by the amount that was requested.	Echidna	**Passed**

`LetterOfCredit.sol`. The `LetterOfCredit.sol` file holds the `LetterOfCredit` contract, which allows creditors to create LOCs that can then be redeemed by beneficiaries. Additional capabilities, such as extending an LOC or converting an LOC, are also provided. We used Echidna to test the creation and redemption of converted LOCs.

Property	Tool	Result
The `createLOC` function should correctly set the creator address, beneficiary address, expiration timestamp, collateral factor, liquidation incentive, collateral vault address, collateral token address, collateral token amount, claimable collateral token amount, credited token address, and credited token amount for converted LOCs.	Echidna	**Passed**
The `createLOC` function should increment the global `locNonce`.	Echidna	**Passed**
The `redeemLOC` function should update the credited token amount for a converted LOC by the amount that is passed in by the sender.	Echidna	**Passed**

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	The contracts in scope use compiler versions (>= 0.8.18) that include arithmetic checks by default and do not use unchecked blocks. Although there may be complexities associated with the mathematical expressions used in price computations, the provided examples and documentation are adequate for comprehending the underlying logic. We also recommend conducting thorough testing of these calculations using automated techniques, such as stateless and stateful fuzzing, to ensure their accuracy and resilience in various scenarios.	**Moderate**
Auditing	All the functions in the DeFi protocol's contracts emit sufficient events to help Acronym Foundation detect unexpected behavior. After each state-changing operation and critical operation, a corresponding event is emitted, ensuring transparency and facilitating the identification of potential issues. Moreover, we did not discover any instances of redundant or misleading event parameters, further affirming the accuracy and reliability of the emitted events. Recommendations to develop an incident response plan can be found in appendix C.	**Satisfactory**
Authentication / Access Controls	All the contracts in the systems have implemented proper access controls for privileged functionalities. Each functionality within the contracts is equipped with appropriate authentication checks, ensuring that only authorized entities can access and modify these functions. In the event of a failed authentication check, the contracts are designed to revert the operation, safeguarding the system's integrity. Additionally, ANVIL uses OpenZeppelin's `ecrecover` library for signature verification, which incorporates checks for invalid signatures. We recommend ANVIL continue to monitor	**Satisfactory**

	security vulnerabilities reported in OpenZeppelin to ensure that vulnerability and bug fixes upstream are patched as quickly as possible. Additionally, it would be beneficial to create user-facing documentation explicitly stating which approved parties are authorized to access each function.	
Complexity Management	The system architecture and the interactions among its components are simple to follow. However, the system involves a substantial amount of logic related to various operations such as collateral claiming, collateral modification, LOC conversion and redemption, and the computation of token pair prices. Although the code is adequately documented with examples and inline comments, it would be beneficial to provide additional inline context specifically for the LOC redemption functionality to detail all possible scenarios for both full and partial redemptions. This would further enhance the clarity and understanding of the codebase.	**Moderate**
Decentralization	Per the documentation, Acronym Foundation will use a multisignature wallet as the initial owner of the contracts. Later, the owner will be updated to a decentralized governance structure. User funds can never be locked by the owner, allowing the user to enter and exit the system at will. Additionally, the owner cannot interfere with any individual LOC to steal funds.	**Moderate**
Documentation	Acronym Foundation's DeFi protocol has excellent documentation that thoroughly explains its design methodology, system actors, threat modeling considerations, and runbooks for each operation. Inline code comments are descriptive and detailed, providing valuable insights. While a few minor comments may be slightly misleading, the majority of the documentation is thorough and effectively clarifies the functionality of the code.	**Satisfactory**
Low-Level Manipulation	The code does not use low-level manipulation. The contracts that do use it are dependencies and were not reviewed for correctness.	**Not Applicable**
Testing and Verification	The test suite has adequate coverage for common happy paths. However, it has limitations in identifying issues	**Moderate**

	related to partial redemptions, which can result in theft of funds (TOB-ANVIL-4). The test suite also overlooks issues related to LOC liquidations (TOB-ANVIL-1, TOB-ANVIL-5, and TOB-ANVIL-7) and pricing computation (TOB-ANVIL-6). We recommend expanding the test suite to uncover any unforeseen behaviors within the contracts. Additionally, integrating a more rigorous fuzzing campaign using Echidna will aid developers in detecting and resolving issues during the development phase.	
Transaction Ordering	At the time of writing, there are no transaction ordering risks inherent to the system itself. There will be races to liquidate unhealthy LOCs, but this is expected system behavior. We recommend that the ANVIL team update the user-facing documentation to highlight how liquidations are used to maintain the expected behavior of the protocol.	**Satisfactory**

Summary of Findings

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

ID	Title	Type	Severity
1	Partial redemptions may prevent insolvent LOCs from being liquidatable	Data Validation	High
2	LOC validity may last longer than maxLocDurationSeconds	Data Validation	Low
3	Lack of zero-value checks in setter functions	Data Validation	Informational
4	Infinite partial redemptions for manually converted LOCs	Data Validation	High
5	Partial redemptions can cause unexpected liquidations	Data Validation	High
6	Incorrect price reporting may cause unexpected liquidations	Data Validation	High
7	LOCs may be immediately liquidatable	Data Validation	Medium

Detailed Findings

1. Partial redemptions may prevent insolvent LOCs from being liquidatable

Severity: **High**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-ANVIL-1
Target: `contracts/LetterOfCredit.sol`	

Description

The amount of transferable collateral is not updated during partial redemptions, which could prevent LOCs from being liquidatable during times of insolvency.

Partial redemptions allow the beneficiary to redeem part of the credited amount while still maintaining the LOC. For unconverted LOCs (uLOCs), a partial redemption requires that some collateral be transferred (claimed) to the `LetterOfCredit` contract and then converted into the credited token, either via a third-party liquidator or with a direct token transfer. Claimable collateral is the amount of collateral backing the LOC after protocol fees have been accounted for.

After the partial redemption, the amount of available credit, the claimable collateral, and the total amount of collateral are the core state variables that must be updated. The updates to the collateral values are done via the `_markLOCPartiallyLiquidated` function (figure 1.1).

```
function _markLOCPartiallyLiquidated(
    uint96 _locId,
    address _initiatorAddress,
    address _liquidatorAddress,
    LOC memory _loc,
    LiquidationContext memory _liquidationContext
) private {
    LOC storage storedLoc = locs[_locId];

    storedLoc.collateralTokenAmount =
        _loc.collateralTokenAmount -
        _liquidationContext.collateralToClaimAndSendLiquidator;

    emit LOCPartiallyLiquidated(
        _locId,
        _initiatorAddress,
        _liquidatorAddress,
```

```
        _liquidationContext.liquidationAmount,
        _liquidationContext.liquidatorFeeAmount,
        _liquidationContext.creditedTokenAmountToReceive
    );
}
```

Figure 1.1: The `_markLOCPartiallyLiquidated` function fails to update the amount of claimable collateral. (`LetterOfCredit._markLOCPartiallyLiquidated#L1001-L1022`)

However, this function updates only the total amount of collateral (highlighted in figure 1.1), not the amount of claimable collateral backing the LOC. This creates issues since, if the LOC becomes insolvent, the operation to claim the necessary amount of collateral from the `Collateral` vault contract will fail, leaving the LOC in a continued state of insolvency. As shown in figure 1.2, the `_calculateLiquidationContext` function will use the non-updated `_loc.claimableCollateral` variable to determine how much should be claimed from the vault. This value will be larger than what is claimable, causing the line highlighted in figure 1.3 to revert.

```
function _calculateLiquidationContext(
    LOC memory _loc,
    uint256 _requiredCreditedAmount,
    bytes memory _oraclePriceUpdate
) private returns (LiquidationContext memory) {
    Pricing.OraclePrice memory price;
    if (_oraclePriceUpdate.length > 0) {
        price = priceOracle.updatePrice{value: msg.value}(_oraclePriceUpdate);
    } else {
        price = priceOracle.getPrice(_loc.collateralTokenAddress,
_loc.creditedTokenAddress);
    }
    _validatePricePublishTime(uint32(price.publishTime));

    /*** Determine if this LOC is insolvent, and if so, adjust credited amount to
receive. ***/
    {
        uint256 claimableCollateralInCreditedToken =
Pricing.collateralAmountInCreditedToken(
            _loc.claimableCollateral,
            price
        );
        if (claimableCollateralInCreditedToken == 0) revert
CollateralAmountInCreditedTokenZero();

        uint256 maxCreditedTokenAmountToReceive = _percentageOfAmount(
            10_000 - _loc.liquidatorIncentiveBasisPoints,
            claimableCollateralInCreditedToken
        );
        if (maxCreditedTokenAmountToReceive <= _loc.creditedTokenAmount) {
            if (_requiredCreditedAmount != _loc.creditedTokenAmount) revert
PartialRedeemInsolvent();
```

```
            // This means that the LOC is insolvent, meaning the collateral is not
enough to pay all fees and LOC face value.
            // The liquidator and protocol will still get their full fees, but the
beneficiary will not get LOC face value.
            // This should never happen, but if somehow it does, the beneficiary
should receive as much as possible.

            uint256 claimableCollateral = _loc.claimableCollateral;

            uint256 fee = _percentageOfAmount(_loc.liquidatorIncentiveBasisPoints,
claimableCollateral);
            return
                LiquidationContext(
                    true,
                    maxCreditedTokenAmountToReceive,
                    claimableCollateral - fee,
                    fee,
                    claimableCollateral
                );
        }
    }
    [...]
}
```

Figure 1.2: The `_calculateLiquidationContext` function uses the incorrect value for the amount of claimable collateral. (`LetterOfCredit._calculateLiquidationContext#L1368-L1435`)

```
function claimCollateral(
    uint96 _collateralId,
    uint256 _amountToReceive,
    address _toAddress,
    bool _releaseRemainder
) external returns (uint256 _remainingReservedCollateral, uint256
_remainingClaimableCollateral) {
    if (_toAddress == address(0)) revert InvalidTargetAddress(_toAddress);
    CollateralReservation memory collateral = collateralReservations[_collateralId];
    if (msg.sender != collateral.collateralizableContract) revert
Unauthorized(msg.sender);

    uint256 amountWithFee = (_amountToReceive * (10_000 +
collateral.feeBasisPoints)) / 10_000;

    if (collateral.tokenAmount < amountWithFee) revert
InsufficientFunds(amountWithFee, collateral.tokenAmount);

    [...]
    IERC20(collateral.tokenAddress).safeTransfer(_toAddress, _amountToReceive);
}
```

Figure 1.3: The `claimCollateral` function will revert since the amount to claim is more than what is available. (`Collateral.claimCollateral#L286-333`)

Exploit Scenario

Alice, the beneficiary, performs a partial redemption of her LOC. After a certain period of time, the LOC becomes insolvent due to a drop in the value of the collateral token backing the LOC. Thus, she attempts to fully redeem the LOC. However, since the amount of claimable collateral was not updated during the first partial redemption, the token transfer from the vault fails, which prevents her from receiving her credited tokens.

Recommendations

Short term, modify the `_markLOCPartiallyLiquidated` function to update the amount of claimable collateral available for the LOC.

Long term, integrate dynamic fuzz testing to ensure that all critical function- and system-level invariants are upheld.

2. LOC validity may last longer than maxLocDurationSeconds

Severity: **Low**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-ANVIL-2
Target: `contracts/LetterOfCredit.sol`	

Description

An LOC that is created from a previously expired one allows the creator to specify an expiration date that is more than the maximum duration allowed.

Users can create a new LOC using the ID of a previously expired LOC as a more efficient way to reuse collateral. This is done via the `createLOCFromExpired` function (figure 2.1). One of the arguments provided to the function is the new `_expirationTimestamp`. The only data validation performed on this argument is to ensure that it is greater than the current `block.timestamp` (figure 2.1).

```
function createLOCFromExpired(
    uint96 _locId,
    address _beneficiary,
    uint32 _expirationTimestamp,
    bytes memory _oraclePriceUpdate
) external payable refundExcess nonReentrant {
    LOC memory loc = locs[_locId];
    uint256 creditedTokenAmount = loc.creditedTokenAmount;

    if (creditedTokenAmount == 0) revert LOCNotFound(_locId);
    if (msg.sender != loc.creator) revert AddressUnauthorizedForLOC(msg.sender,
_locId);
    if (_expirationTimestamp <= block.timestamp) revert LOCExpired(0,
_expirationTimestamp);
    if (loc.expirationTimestamp > block.timestamp) revert LOCNotExpired(_locId);

    [...]

    _persistAndEmitNewLOCCreated(
        loc.collateralId,
        msg.sender,
        _beneficiary,
        collateralTokenAddress,
        loc.collateralTokenAmount,
        loc.claimableCollateral,
```

```
        creditedTokenAddress,
        creditedTokenAmount,
        _expirationTimestamp
    );
}
```

Figure 2.1: The `createLOCFromExpired` function fails to ensure that the new `_expirationTimestamp` does not create an LOC that is valid for more than `maxLocDurationSeconds`. (`LetterOfCredit.createLOCFromExpired#L348-399`)

However, there is no validation to ensure that the difference between `_expirationTimestamp` and `block.timestamp` is less than or equal to the `maxLocDurationSeconds` global state variable, which is a violation of a system invariant.

Exploit Scenario

Eve calls `createLOCFromExpired` with a very large `_expirationTimestamp`, which allows the LOC to last for much longer than `maxLocDurationSeconds`.

Recommendations

Short term, add a check to the `createLOCFromExpired` function to ensure that the difference between `_expirationTimestamp` and `block.timestamp` is less than or equal to `maxLocDurationSeconds`.

Long term, improve unit test coverage to identify edge cases like this.

3. Lack of zero-value checks in setter functions

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-ANVIL-3
Target: `contracts/Collateral.sol`	

Description

Certain functions fail to validate incoming arguments, so callers of these functions could mistakenly set important state variables to a zero value, misconfiguring the system.

For example, the `upsertCollateralizableContractApproval` function in the `Collateral` contract sets approval for a contract address, `_contractAddress`, in the `collateralizableContracts` mapping without checking whether the `_contractAddress` is zero. This may result in undefined behavior in the system.

```
1    function upsertCollateralizableContractApproval(address _contractAddress, bool
                                                    _approved) external onlyOwner {
2        collateralizableContracts[_contractAddress] = _approved;
3        emit CollateralizableContractApprovalUpdated(_approved, _contractAddress);
4    }
```

Figure 3.1: The `upsertCollateralizableContractApproval` function does not check `_contractAddress` for a zero-value. (`Collateral.upsertCollateralizableContractApproval#L499-L503`)

The following function also fails to perform zero-value checks:

- The `upsertAccountCollateralizableContractApproval` function in `contracts/Collateral.sol`

Exploit Scenario

Alice deploys a new version of the `Collateral` contract. When she invokes `upsertCollateralizableContractApproval` to set approval for `_contractAddress`, she accidentally enters a zero value, thereby misconfiguring the system.

Recommendations

Short term, add zero-value checks to all function arguments to ensure that callers cannot set incorrect values and misconfigure the system.

Long term, use the Slither static analyzer to catch common issues such as this one. Consider integrating a Slither scan into the project's CI pipeline, pre-commit hooks, or build scripts.

4. Infinite partial redemptions for manually converted LOCs

Severity: **High**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-ANVIL-4
Target: `contracts/LetterOfCredit.sol`	

Description

In cases of partial redemptions, the credited token amount is not updated for manually converted LOCs, which leads to infinite partial redemptions and drains funds from the `LetterOfCredit` contract.

Partial redemptions allow the beneficiary to redeem part of the credited amount while still maintaining the LOC. During partial redemption, part of the credited token amount is transferred to the beneficiary, so the critical storage variables for the LOC, such as `collateralTokenAmount` and `creditedTokenAmount`, should be updated to reflect the remaining collateral and credited token balance. However, in certain cases, the `LetterOfCredit` contract fails to update those variables, allowing infinite partial redemptions by the beneficiary.

The `createLOC` function in the `LetterOfCredit` contract creates an unconverted LOC that can be converted using the `convertLOC` function before the beneficiary partially redeems the LOC. After the conversion, the `collateralId` of the LOC is set to `0`, as shown in figure 4.1.

```
1    function _markLOCConverted(
2        uint96 _locId,
3        address _initiatorAddress,
4        address _liquidatorAddress,
5        LOC memory _loc,
6        LiquidationContext memory _liquidationContext
7    ) private {
8        LOC storage storedLoc = locs[_locId];
9        ...
10       storedLoc.collateralId = 0;
11
12       emit LOCConverted(
13           _locId,
14           _initiatorAddress,
15           _liquidatorAddress,
16           _liquidationContext.liquidationAmount,
```

```
            _liquidationContext.liquidatorFeeAmount,
            _liquidationContext.creditedTokenAmountToReceive
        );
    }
```

Figure 4.1: The `_markLOCConverted` function sets `collateralId` to `0`. (`LetterOfCredit._markLOCConverted#L968-991`)

As this is a manually converted LOC with a `collateralId` of `0`, the `else` branch of the conditional block is triggered during partial redemption, as highlighted in figure 4.2. However, after the required redemption amount is transferred to the beneficiary, the remaining collateral token amount and the credited token amount are not adjusted to reflect the updated token amounts. Consequently, this allows the beneficiary to repeatedly perform partial redemptions on the LOC until the `LetterOfCredit` contract exhausts its token balance.

```
 1    function _redeemLOC(
 2        uint96 _locId,
 3        LOC memory _loc,
 4        uint256 _creditedTokenAmountToRedeem,
 5        address _destinationAddress,
 6        address _iLiquidatorToUse,
 7        bytes calldata _oraclePriceUpdate
 8    ) private {
 9        ...
10        bool isPartialRedeem = _creditedTokenAmountToRedeem !=
                                 _loc.creditedTokenAmount;
11        ...
12        if (_loc.collateralTokenAddress != _loc.creditedTokenAddress) {
13            ...
14        } else if (_loc.collateralId != 0) {
15            ...
16        } else {
17            // This means the LOC was already manually converted,
                 so the funds are held by this contract. Send to beneficiary.
18            // Note: no collateral is used in redeem, so those variables are not
19            IERC20(_loc.creditedTokenAddress).safeTransfer(_destinationAddress,
                                                 _creditedTokenAmountToRedeem);
20        }
21        ...
22    }
```

Figure 4.2: The `_redeemLOC` function does not update the credited token amount in case of partial redemption. (`LetterOfCredit._redeemLOC#L1224-1228`)

Exploit Scenario

Alice creates an unconverted LOC by calling the `createLOC` function in the `LetterOfCredit` contract, keeping Bob as the beneficiary. Alice then invokes the `convertLOC` function to convert the newly created LOC. Later, Bob partially redeems the LOC through the `_redeemLOC` function. However, `_redeemLOC` does not update the

remaining credit token amount and collateral amount after the partial redemption. As a result, Bob invokes the `_redeemLOC` function several times to redeem the LOC until the `LetterOfCredit` contract is out of credited tokens.

Recommendations

Short term, modify the `_redeemLOC` function to update the remaining credited token amount and the collateral token amount in case of partial redemptions.

Long term, incorporate dynamic testing tools like Echidna to verify the integrity of all system-level invariants.

5. Partial redemptions can cause unexpected liquidations

Severity: **High**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-ANVIL-5
Target: `contracts/LetterOfCredit.sol`	

Description

The amount of credited tokens available for redemption is not updated during partial redemptions, which would cause LOCs to enter an unhealthy state and allow them to then be unexpectedly liquidated.

Similar to TOB-ANVIL-1, partial redemptions must update the amount of credited tokens, the claimable collateral, and the total amount of collateral for the LOC. These updates must be done via the `_markLOCPartiallyLiquidated` function (figure 5.1).

```
function _markLOCPartiallyLiquidated(
    uint96 _locId,
    address _initiatorAddress,
    address _liquidatorAddress,
    LOC memory _loc,
    LiquidationContext memory _liquidationContext
) private {
    LOC storage storedLoc = locs[_locId];

    storedLoc.collateralTokenAmount =
        _loc.collateralTokenAmount -
        _liquidationContext.collateralToClaimAndSendLiquidator;

    emit LOCPartiallyLiquidated(
        _locId,
        _initiatorAddress,
        _liquidatorAddress,
        _liquidationContext.liquidationAmount,
        _liquidationContext.liquidatorFeeAmount,
        _liquidationContext.creditedTokenAmountToReceive
    );
}
```

Figure 5.1: The `_markLOCPartiallyLiquidated` function does not update the credited token amount. (`LetterOfCredit._markLOCPartiallyLiquidated#L1001-L1022`)

However, the function does not update the credited token amount, so the LOC will now become unhealthy (figure 5.2).

```
function _calculateLiquidationContext(
    LOC memory _loc,
    uint256 _requiredCreditedAmount,
    bytes memory _oraclePriceUpdate
) private returns (LiquidationContext memory) {
    Pricing.OraclePrice memory price;
    if (_oraclePriceUpdate.length > 0) {
        price = priceOracle.updatePrice{value: msg.value}(_oraclePriceUpdate);
    } else {
        price = priceOracle.getPrice(_loc.collateralTokenAddress,
_loc.creditedTokenAddress);
    }
    _validatePricePublishTime(uint32(price.publishTime));

    /*** Determine if this LOC is insolvent, and if so, adjust credited amount to
receive. ***/
    {
        [...]
    }

    /*** LOC is not insolvent, so calculate liquidation amounts, leaving collateral
to be received untouched. ***/

    uint256 collateralAmountInCreditedToken =
Pricing.collateralAmountInCreditedToken(
        _loc.collateralTokenAmount,
        price
    );

    uint256 liquidationAmount = (_loc.collateralTokenAmount *
_requiredCreditedAmount) /
        collateralAmountInCreditedToken;
    uint256 liquidatorFeeAmount =
_percentageOfAmount(_loc.liquidatorIncentiveBasisPoints, liquidationAmount);

    // NB: Truncation is fine because we're checking >= for unhealthy below
    uint256 collateralFactorBasisPoints = (_loc.creditedTokenAmount * 10_000) /
collateralAmountInCreditedToken;

    return
        LiquidationContext(
            collateralFactorBasisPoints >= _loc.collateralFactorBasisPoints,
            _requiredCreditedAmount,
            liquidationAmount,
            liquidatorFeeAmount,
            liquidationAmount + liquidatorFeeAmount
        );
}
```

Figure 5.2: The `_calculateLiquidationContext` function uses the incorrect value for the amount of credited tokens. (`LetterOfCredit._calculateLiquidationContext#L1368-L1435`)

Since the LOC is now unhealthy, anyone can call the `convertLOC` function, liquidate the collateral, and receive the liquidation incentive. If the credited token amount was correctly accounted for, the LOC would remain in a healthy state and only the creator could call `convertLOC`.

The fix for TOB-ANVIL-1 and this issue (see Recommendations below) triggers another bug in the `_redeemLOC` function. The conditional logic in figure 5.3 incorrectly assesses a complete redemption, a partial redemption, or an insolvent LOC. This logic must also be updated.

```
function _redeemLOC(
    uint96 _locId,
    LOC memory _loc,
    uint256 _creditedTokenAmountToRedeem,
    address _destinationAddress,
    address _iLiquidatorToUse,
    bytes calldata _oraclePriceUpdate
) private {
    if (_loc.creditedTokenAmount == 0) revert LOCNotFound(_locId);
    if (_loc.expirationTimestamp <= block.timestamp) revert LOCExpired(_locId,
_loc.expirationTimestamp);
    if (_creditedTokenAmountToRedeem == 0 || _creditedTokenAmountToRedeem >
_loc.creditedTokenAmount)
        revert InvalidRedeemAmount(_creditedTokenAmountToRedeem,
_loc.creditedTokenAmount);
    if (_destinationAddress == address(0)) revert InvalidZeroAddress();

    bool isPartialRedeem = _creditedTokenAmountToRedeem != _loc.creditedTokenAmount;

    uint256 collateralUsed = 0;
    uint256 claimableCollateralUsed = 0;
    uint256 redeemedAmount = _creditedTokenAmountToRedeem;
    if (_loc.collateralTokenAddress != _loc.creditedTokenAddress) {
        // Needs to be converted, so convert.
        // Note: LOC collateral is updated in storage as part of this operation.
        (collateralUsed, claimableCollateralUsed) = _liquidateLOCCollateral(
            _locId,
            _creditedTokenAmountToRedeem,
            _iLiquidatorToUse,
            msg.sender,
            _oraclePriceUpdate,
            true
        );

        // NB: Liquidation will update the creditedTokenAmount with the portion it
was able to get if insolvent.
        uint256 creditedAmountAfterLiquidation = locs[_locId].creditedTokenAmount;
        if (creditedAmountAfterLiquidation == _loc.creditedTokenAmount) {
            // LOC is not insolvent.
            if (isPartialRedeem) {
                locs[_locId].creditedTokenAmount = creditedAmountAfterLiquidation -
```

```
_creditedTokenAmountToRedeem;
            }
            IERC20(_loc.creditedTokenAddress).safeTransfer(_destinationAddress,
_creditedTokenAmountToRedeem);
        } else {
            // LOC is insolvent. The contract does not allow partial redemption of
insolvent LOCs, so send remaining LOC value.
            IERC20(_loc.creditedTokenAddress).safeTransfer(_destinationAddress,
creditedAmountAfterLiquidation);
            redeemedAmount = creditedAmountAfterLiquidation;
        }
    } else if (_loc.collateralId != 0) {
        [...]
    } else {
        [...]
    }

    if (!isPartialRedeem) {
        delete locs[_locId];
    }

    emit LOCRedeemed(_locId, _destinationAddress, redeemedAmount, collateralUsed,
claimableCollateralUsed);
}
```

Figure 5.3: The `_redeemLOC` function contains a bug that is triggered after the fix is made for this issue and TOB-ANVIL-1. (`LetterOfCredit._redeemLOC#L1159-L1235`)

Exploit Scenario

Alice, the beneficiary of a uLOC, performs a partial redemption of her credit. However, since the amount of credited tokens that Alice is due was not decremented during the redemption process, the LOC is now in an unhealthy state. Eve immediately liquidates the position.

Recommendations

Short term, modify the `_markLOCPartiallyLiquidated` function to update the amount of credited tokens available for the LOC.

Long term, thoroughly review the partial redemption code paths to ensure that all the necessary state changes are made and that they do not violate any critical invariants. Additionally, incorporate dynamic testing tools like Echidna to verify the integrity of all function- and system-level invariants.

6. Incorrect price reporting may cause unexpected liquidations

Severity: **High**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-ANVIL-6
Target: `contracts/PythPriceOracle.sol`	

Description

Due to incorrect data validation when updating the available Pyth price feeds, prices may be reported incorrectly. This would cause uLOCs to reach an unhealthy state and allow an attacker to liquidate them.

The `PythPriceOracle` contract owner can update the tokens and price feeds that the protocol supports by calling the `_upsertPriceFeedIdsAsOwner` function (figure 6.1). This function would be called, for example, if the ANVIL team decides to support a new token or disable one.

```
function _upsertPriceFeedIdsAsOwner(address[] memory _tokenAddresses, bytes32[]
memory _priceFeedIds) private {
    if (_tokenAddresses.length != _priceFeedIds.length)
        revert RelatedArraysLengthMismatch(_tokenAddresses.length,
_priceFeedIds.length);

    for (uint256 i = 0; i < _tokenAddresses.length; i++) {
        TokenInfo storage tokenInfo = addressToTokenInfo[_tokenAddresses[i]];
        bytes32 oldPriceFeedId = tokenInfo.priceFeedId;
        tokenInfo.priceFeedId = _priceFeedIds[i];
        if (_priceFeedIds[0] == bytes32(0)) {
            tokenInfo.decimals = 0;
        } else {
            uint8 decimals = IERC20Metadata(_tokenAddresses[i]).decimals();
            tokenInfo.decimals = decimals;
        }
        emit PriceFeedUpdated(_tokenAddresses[i], oldPriceFeedId, _priceFeedIds[i]);
    }
}
```

Figure 6.1: The `_upsertPriceFeedIdsAsOwner` function incorrectly disables price feeds. (`PythPriceOracle._upsertPriceFeedIdsAsOwner#L263-L279`)

However, in the highlighted portion of figure 6.1, if the first price feed ID in the `_priceFeedIds` array is `bytes(0)`, which causes the associated token to be effectively disabled, all `TokenInfo` objects will have their decimal value set to `0`. Thus, disabling the

first supported token causes incorrect price information to be set for the rest of the tokens in the array. This causes an issue when a caller calls the `getPrice` function (figure 6.2).

```
function getPrice(
    address _inputTokenAddress,
    address _outputTokenAddress
) external view returns (Pricing.OraclePrice memory _price) {
    TokenInfo memory inputTokenInfo =
_fetchAndValidateTokenInfo(_inputTokenAddress);
    TokenInfo memory outputTokenInfo =
_fetchAndValidateTokenInfo(_outputTokenAddress);

    // NB: getPriceUnsafe because callers of this function do their own recency
checks.
    // Get token USD prices & ensure positive
    PythStructs.Price memory inputUsdPrice =
pythContract.getPriceUnsafe(inputTokenInfo.priceFeedId);
    if (inputUsdPrice.price <= 0) revert InvalidOraclePrice(_inputTokenAddress,
inputUsdPrice.price);

    PythStructs.Price memory outputUsdPrice =
pythContract.getPriceUnsafe(outputTokenInfo.priceFeedId);
    if (outputUsdPrice.price <= 0) revert InvalidOraclePrice(_outputTokenAddress,
outputUsdPrice.price);

    // NB: outputPerUnitInput = inputPrice * 10**(inputDecimals - outputExponent +
inputExponent + 1)  / outputPrice; exponent: (- outputDecimals - 1)
    int32 expSum = inputUsdPrice.expo - outputUsdPrice.expo +
int32(uint32(inputTokenInfo.decimals)) + 1;

    // NB: target 10 digits of precision minimum
    int32 precisionBufferExponent = 10 -
        expSum -
        int32(int256(Math.log10(uint256(uint64(inputUsdPrice.price))))) +
        int32(int256(Math.log10(uint256(uint64(outputUsdPrice.price)))));

    if (precisionBufferExponent < 0) {
        precisionBufferExponent = 0;
    }

    uint256 precisionBuffer = 10 ** uint256(uint32(precisionBufferExponent));

    if (expSum >= 0) {
        _price.price =
            ((uint256(uint64(inputUsdPrice.price)) * 10 ** uint256(uint32(expSum)))
* precisionBuffer) /
            uint256(uint64(outputUsdPrice.price));
    } else {
        _price.price = ((uint256(uint64(inputUsdPrice.price)) * precisionBuffer) /
            uint256(uint64(outputUsdPrice.price)) /
            10 ** uint256(uint32(-expSum)));
    }
```

```
    _price.exponent = -(int32(uint32(outputTokenInfo.decimals))) - 1 -
precisionBufferExponent;

    if (inputUsdPrice.publishTime < outputUsdPrice.publishTime) {
        _price.publishTime = inputUsdPrice.publishTime;
    } else {
        _price.publishTime = outputUsdPrice.publishTime;
    }
}
```

Figure 6.2: The `getPrice` function will report incorrect information due to the input and output tokens having zero decimals. (`PythPriceOracle.getPrice#L140-L186`)

As highlighted in figure 6.2, since the decimal values will be set to `0` for the input and output tokens, the final reported price will be magnitudes away from the expected price. The incorrectly reported price may unexpectedly cause a healthy LOC to become an unhealthy one. Thus, a malicious user can call `LetterOfCredit.convertLOC`, liquidate the LOC, and make a profit.

Exploit Scenario

Alice, the owner of the `PythPriceOracle` contract, decides to disable TKNA and update the price feed ID for TKNB. When she calls `upsertPriceFeedIds`, TKNA is at index zero of the `_tokenAddresses` array and `bytes(0)` is set as the associated price feed ID in the `_priceFeedIds` array. This causes TKNB's `TokenInfo.decimals` value to be set to `0` and leads to an incorrect price being reported for all LOCs that use TKNB as the credited or collateral token. Eve notices the incorrect price, searches for LOCs that are vulnerable, and liquidates all of them to make a profit.

Recommendations

Short term, update the conditional logic to `if (_priceFeedIds[i] == bytes32(0))`.

Long term, improve unit test branch coverage to ensure that all conditionals are thoroughly tested with extensive post-condition checks.

7. LOCs may be immediately liquidatable

Severity: **Medium**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-ANVIL-7
Target: `contracts/LetterOfCredit.sol`	

Description

If the creation collateral factor (CCF) and the liquidation collateral factor (LCF) are equal, then LOCs that have the same collateral factor (CF) are immediately liquidatable.

During the creation of a collateral and credit asset pair, the owner of the `LetterOfCredit` contract will specify the CCF and LCF at which the LOC can be created and liquidated, respectively (figure 7.1). In the highlighted portion of the figure, the LCF must be *greater than or equal to* the CCF.

```
function _upsertCollateralFactorsAsOwner(AssetPairCollateralFactor[] memory
_assetPairCollateralFactors) private {
    for (uint256 i = 0; i < _assetPairCollateralFactors.length; i++) {
        AssetPairCollateralFactor memory apcf = _assetPairCollateralFactors[i];
        CollateralFactor memory cf = apcf.collateralFactor;

        uint16 liquidatorIncentiveBasisPoints = cf.liquidatorIncentiveBasisPoints;
        if (liquidatorIncentiveBasisPoints > 10_000) revert
InvalidBasisPointValue(liquidatorIncentiveBasisPoints);

        uint16 creationCFBasisPoints = cf.creationCollateralFactorBasisPoints;
        if (creationCFBasisPoints > 10_000) revert
InvalidBasisPointValue(creationCFBasisPoints);

        uint16 liquidationCFBasisPoints = cf.collateralFactorBasisPoints;
        if (liquidationCFBasisPoints < creationCFBasisPoints)
            revert InvalidCollateralFactor(liquidationCFBasisPoints,
creationCFBasisPoints);

        [...]
    }
}
```

Figure 7.1: The `_upsertCollateralFactorsAsOwner` function asserts that the LCF is greater than or equal to the CCF.
(`LetterOfCredit._upsertCollateralFactorsAsOwner#L1454-L1486`)

Thus, a user can create an LOC that has a CF that is equal to both the CCF and LCF. During LOC creation, the CF must be *less than or equal to* the CCF. Thus, if the CF is equal to the

CCF, it is also equal to the LCF. This edge case, however, immediately allows anyone to liquidate the LOC because the `_calculateLiquidationContext` function specifies that an LOC that has a CF equal to the LCF is unhealthy (figure 7.2). This would allow anyone to call the `convertLOC` function and liquidate the position.

```
function _calculateLiquidationContext(
    LOC memory _loc,
    uint256 _requiredCreditedAmount,
    bytes memory _oraclePriceUpdate
) private returns (LiquidationContext memory) {
    Pricing.OraclePrice memory price;
    if (_oraclePriceUpdate.length > 0) {
        price = priceOracle.updatePrice{value: msg.value}(_oraclePriceUpdate);
    } else {
        price = priceOracle.getPrice(_loc.collateralTokenAddress,
_loc.creditedTokenAddress);
    }
    _validatePricePublishTime(uint32(price.publishTime));

    /*** Determine if this LOC is insolvent, and if so, adjust credited amount to
receive. ***/
    {
        [...]
    }

    /*** LOC is not insolvent, so calculate liquidation amounts, leaving collateral
to be received untouched. ***/

    uint256 collateralAmountInCreditedToken =
Pricing.collateralAmountInCreditedToken(
        _loc.collateralTokenAmount,
        price
    );

    uint256 liquidationAmount = (_loc.collateralTokenAmount *
_requiredCreditedAmount) /
        collateralAmountInCreditedToken;
    uint256 liquidatorFeeAmount =
_percentageOfAmount(_loc.liquidatorIncentiveBasisPoints, liquidationAmount);

    // NB: Truncation is fine because we're checking >= for unhealthy below
    uint256 collateralFactorBasisPoints = (_loc.creditedTokenAmount * 10_000) /
collateralAmountInCreditedToken;

    return
        LiquidationContext(
            collateralFactorBasisPoints >= _loc.collateralFactorBasisPoints,
            _requiredCreditedAmount,
```

```
            liquidationAmount,
            liquidatorFeeAmount,
            liquidationAmount + liquidatorFeeAmount
        );
}
```

Figure 7.2: The `_calculateLiquidationContext` function will identify an LOC with a CF that is greater than or equal to the LCF as unhealthy. (`LetterOfCredit._calculateLiquidationContext#L1368-L1435`

Exploit Scenario

Alice creates an asset pair of collateral TKNA and credit TKNB where the LCF and CCF are equal. Bob subsequently creates an LOC with a CF that is equal to the CCF, which in turn makes it equal to the LCF. Bob's position is considered unhealthy. Eve immediately liquidates the position.

Recommendations

Short term, update the `_upsertCollateralFactorsAsOwner` function so that the `liquidationCFBasisPoints` variable must be strictly *greater than* the `creationCFBasisPoints` variable. This will prevent the LOC from being immediately liquidatable.

Long term, identify all system-level invariants that must be upheld and use dynamic fuzz testing to validate them.

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
Session Management	Improper identification of authenticated users
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
Front-Running Resistance	The system's resistance to front-running 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. Incident Response Recommendations

This section provides recommendations on formulating an incident response plan.

- **Identify the parties (either specific people or roles) responsible for implementing the mitigations when an issue occurs (e.g., deploying smart contracts, pausing contracts, upgrading the front end, etc.).**
- **Document internal processes for addressing situations in which a deployed remedy does not work or introduces a new bug.**
    - Consider documenting a plan of action for handling failed remediations.
- **Clearly describe the intended contract deployment process.**
- **Outline the circumstances under which Acronym will compensate users affected by an issue (if any).**
    - Issues that warrant compensation could include an individual or aggregate loss or a loss resulting from user error, a contract flaw, or a third-party contract flaw.
- **Document how the team plans to stay up to date on new issues that could affect the system; awareness of such issues will inform future development work and help the team secure the deployment toolchain and the external on-chain and off-chain services that the system relies on.**
    - Identify sources of vulnerability news for each language and component used in the system, and subscribe to updates from each source. Consider creating a private Discord channel in which a bot will post the latest vulnerability news; this will provide the team with a way to track all updates in one place. Lastly, consider assigning certain team members to track news about vulnerabilities in specific system components.
- **Determine when the team will seek assistance from external parties (e.g., auditors, affected users, other protocol developers) and how it will onboard them.**
    - Effective remediation of certain issues may require collaboration with external parties.
- **Define contract behavior that would be considered abnormal by off-chain monitoring solutions.**

It is best practice to perform periodic dry runs of scenarios outlined in the incident response plan to find omissions and opportunities for improvement and to develop "muscle memory." Additionally, document the frequency with which the team should perform dry runs of various scenarios, and perform dry runs of more likely scenarios more regularly. Create a template to be filled out with descriptions of any necessary improvements after each dry run.

D. Code Quality Recommendations

The following recommendations are not associated with any specific vulnerabilities. However, they enhance code readability and may prevent the introduction of vulnerabilities in the future.

- Consider removing the (`oldAmount == 0`) check for `_collateralId` existence in the `modifyCollateralReservation` function. This is redundant, as the subsequent line already performs a similar check (`msg.sender != reservation.collateralizableContract`).
- The inline comment while declaring the `creditedTokens` variable in the `LetterOfCredit` contract is stated as `Collateral Token Address => token`. Instead, it should be `Credited token address => token`.

E. 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.

ERC-721 Tokens

ERC-721 Conformity Checks

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

- ❏ **Transfers of tokens to the `0x0` address revert.** Several tokens allow transfers to `0x0` and consider tokens transferred to that address to have been burned; however, the ERC-721 standard requires that such transfers revert.
- ❏ **`safeTransferFrom` functions are implemented with the correct signature.** Several token contracts do not implement these functions. A transfer of NFTs to one of these contracts can result in a loss of assets.
- ❏ **The `name`, `decimals`, and `symbol` functions are present if used.** These functions are optional in the ERC-721 standard and may not be present.
- ❏ **If it is used, `decimals` returns a `uint8(0)`.** Other values are invalid.
- ❏ **The `name` and `symbol` functions can return an empty string.** This behavior is allowed by the standard.
- ❏ **The `ownerOf` function reverts if the `tokenID` is invalid or is set to a token that has already been burned.** The function cannot return `0x0`. This behavior is required by the standard, but it is not always properly implemented.
- ❏ **A transfer of an NFT clears its approvals.** This is required by the standard.
- ❏ **The `tokenID` of an NFT cannot be changed during its lifetime.** This is required by the standard.

Common Risks of the ERC-721 Standard

To mitigate the risks associated with ERC-721 contracts, conduct a manual review of the following conditions:

- ❏ **The `onERC721Received` callback is accounted for.** External calls in the transfer functions can lead to reentrancies, especially when the callback is not explicit (e.g., in `safeMint` calls).
- ❏ **When an NFT is minted, it is safely transferred to a smart contract.** If there is a minting function, it should behave like `safeTransferFrom` and properly handle the minting of new tokens to a smart contract. This will prevent a loss of assets.
- ❏ **The burning of a token clears its approvals.** If there is a burning function, it should clear the token's previous approvals.

F. 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 1 to December 5, 2023, Trail of Bits reviewed the fixes and mitigations implemented by the Acronym team for the issues identified in this report. We reviewed each fix to determine its effectiveness in resolving the associated issue.

In summary, the Acronym team has resolved all seven issues described in this report. For additional information, please see the Detailed Fix Review Results below.

ID	Title	Status
1	Partial redemptions may prevent insolvent LOCs from being liquidatable	Resolved
2	LOC validity may last longer than maxLocDurationSeconds	Resolved
3	Lack of zero-value checks in setter functions	Resolved
4	Infinite partial redemptions for manually converted LOCs	Resolved
5	Partial redemptions can cause unexpected liquidations	Resolved
6	Incorrect price reporting may cause unexpected liquidations	Resolved
7	LOCs may be immediately liquidatable	Resolved

Detailed Fix Review Results

TOB-ANVIL-1: Partial redemptions may prevent insolvent LOCs from being liquidatable
Resolved in commit a321af1d. The `_markLOCPartiallyLiquidated` function was modified to update the amount of claimable collateral available for the LOC.

TOB-ANVIL-2: LOC validity may last longer than maxLocDurationSeconds
Resolved in commit 2c6c2000. The `createLOCFromExpired` function was updated to incorporate a check that ensures the difference between `_expirationTimestamp` and `block.timestamp` is less than or equal to `maxLocDurationSeconds`. If this condition does not hold, the function will trigger a revert operation.

TOB-ANVIL-3: Lack of zero-value checks in setter functions
Resolved in commit 6b773e35. Zero-value checks were added to all relevant setter functions.

TOB-ANVIL-4: Infinite partial redemptions for manually converted LOCs
Resolved in commit a321af1d**.** The `_redeemLOC` function was restructured and modified heavily to enhance readability and eliminate redundant logic related to partial redemptions. Additionally, the remaining credited token amount and the collateral token amount were updated in case of partial redemptions.

TOB-ANVIL-5: Partial redemptions can cause unexpected liquidations
Resolved in commit a321af1d. The logic surrounding partial redemptions was reworked and simplified to enhance readability. Additionally, the `_markLOCPartiallyLiquidated` function was modified to update the amount of credited token available for redemption during partial redemptions.

TOB-ANVIL-6: Incorrect price reporting may cause unexpected liquidations
Resolved in commit 44b3762a. The `_upsertPriceFeedIdsAsOwner` function was modified to fix the conditional logic for updating the price feed.

TOB-ANVIL-7: LOCs may be immediately liquidatable
Resolved in commit 63cd324b. The `_upsertCollateralFactorsAsOwner` function was modified to require the `liquidationCFBasisPoints` variable to be strictly greater than the `creationCFBasisPoints` variable. If this condition is not met, it will trigger a revert operation.

G. 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.