



Sherlock Protocol

Security Assessment

June 29, 2022

Prepared for:

Jack Sanford and Evert Kors

Sherlock

Prepared by: **Simone Monica and Justin Jacob**

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 80+ 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

© 2022 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 Sherlock under the terms of the project statement of work and has been made public at Sherlock'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.

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

About Trail of Bits	1
Notices and Remarks	2
Table of Contents	3
Executive Summary	5
Project Summary	7
Project Goals	8
Project Targets	9
Project Coverage	10
Codebase Maturity Evaluation	11
Summary of Findings	13
Detailed Findings	14
1. Solidity compiler optimizations can be problematic	14
2. Lack of events for critical operations	15
3. Infinite allowances pose security risks	17
4. claimReward function fails in paused system	18
5. TrueFi deposit can revert	19
6. AlphaBetaEqualDepositMaxSplitter can start with a child whose value exceeds the maximum amount	21
7. Lack of two-step process for contract ownership changes	23
8. USDC transfer could fail silently	24
9. Aave strategy could leave funds in the contract when withdrawing	25
Summary of Recommendations	26
A. Vulnerability Categories	27

B. Code Maturity Categories	29
C. Code Quality Recommendations	31

Executive Summary

Engagement Overview

Sherlock engaged Trail of Bits to review the security of its smart contracts. From May 23 to June 3, 2022, a team of two consultants conducted a security review of the client-provided source code, with four person-weeks of effort. Details of the project's timeline, test targets, and coverage are provided in subsequent sections of this report.

Project Scope

Our testing efforts were focused on the identification of flaws that could result in a compromise of confidentiality, integrity, or availability of the target system. We conducted this audit with full knowledge of the target system, including access to the source code and documentation. We performed automated and manual static testing of the target system and its codebase.

Summary of Findings

The audit uncovered a significant flaw that could impact system confidentiality, integrity, or availability. A summary of the finding is provided below.

EXPOSURE ANALYSIS

<i>Severity</i>	<i>Count</i>
High	1
Medium	1
Low	5
Informational	2

CATEGORY BREAKDOWN

<i>Category</i>	<i>Count</i>
Access Controls	1
Auditing and Logging	1
Data Validation	2
Denial of Service	1
Undefined Behavior	4

Notable Findings

Significant flaws that impact system confidentiality, integrity, or availability are listed below.

- **Risks associated with contract ownership transfer process (TOB-SHER-7)**
Contract ownership is transferred in a single step, which prevents the reversal of ownership transfers if the wrong address is inputted. Implementing a two-step process for transferring contract ownership would allow ownership transfers to be reversed if an incorrect address is input. This is an issue inherent in OpenZeppelin contracts.

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:

Simone Monica, Consultant
simone.monica@trailofbits.com

Justin Jacob, Consultant
justin.jacob@trailofbits.com

Project Timeline

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

Date	Event
May 19, 2022	Pre-project kickoff call
May 27, 2022	Status update meeting #1
June 3, 2022	Delivery of report draft
June 3, 2022	Report readout meeting
June 29, 2022	Delivery of final report

Project Goals

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

- Are the access controls applied correctly?
- Is failure possible when depositing or withdrawing funds into or from the strategies?
- Can the strategies lose funds?
- Are the correct checks applied to the tree strategies' actions?
- Can the tree strategy reach an incorrect state?
- Is the whenPaused modifier applied correctly?
- Are any ownership transfers prone to errors?
- Can ERC20 transfers silently fail, causing the system to go into an undefined state?

Project Targets

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

Sherlock V2

Repository	https://github.com/sherlock-protocol/sherlock-v2-core/tree/strategy
Version	6ff8bdac5c186acaa6b5d333596065272758c1e9
Type	Solidity
Platform	Ethereum
Scope	contracts/managers/MasterStrategy.sol contracts/strategy/*

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 their results include the following:

- We reviewed the access controls applied to the individual strategies and the tree strategy. All the possible actions can be made only by the owner of the `masterStrategy` contract or the `sherlockCore` contract. Additionally, we reviewed the safety of contract ownership transfer, which led to [TOB-SHER-7](#).
- We investigated whether errors can occur when individual strategies interact with the corresponding protocol, such as missing checks that could lead to reverting an operation when it should not ([TOB-SHER-5](#)), the correct calculation of the strategy's balance, and possible loss of funds ([TOB-SHER-3](#)).
- We analyzed the correctness of state changes regarding the tree strategy's actions such as replacing or adding a splitter/strategy or removing a strategy. This led to the discovery of [TOB-SHER-6](#).
- We investigated the pausability of the contracts and found that the `whenNotPaused` modifier was incorrectly applied to the `claimReward` function when strategies were paused ([TOB-SHER-4](#)).
- We reviewed the deposit and withdrawal processes from the splitter contract to its respective child contract and discovered [TOB-SHER-8](#).

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 use of <code>solc 0.8.10</code> prevents underflows and overflows. No unchecked blocks are used. The arithmetic operations are simple to understand.	Satisfactory
Auditing	Several critical state changes in the contract do not emit events (TOB-SHER-2). Additionally, it is unclear if the Sherlock team is using a blockchain monitoring system and if they have an incident response plan.	Moderate
Authentication / Access Controls	Appropriate access controls are in place for updating the tree structure. Only a single privileged actor can do this.	Satisfactory
Complexity Management	All contracts' functionalities are simple and easy to understand. The design is well separated, modular, and thoroughly documented.	Satisfactory
Decentralization	The functions in the contracts are designed to be called only by the owner of the Sherlock contract. Currently, the owner is a multi-signature wallet controlled by the Sherlock team; however, a planned transfer of ownership to a DAO will decentralize the protocol's governance.	Moderate
Documentation	The protocol has comprehensive documentation in the form of diagrams and wiki entries. All functions in the codebase have docstrings and comments explaining their purpose. However, we found a few incorrect comments, and the documentation regarding the replacing of the splitters could be improved. Moreover, the user	Moderate

	documentation must be updated with the new tree strategy.	
Front-Running Resistance	We did not find any front-running issues. However, we did not exhaustively check the protocol for front-running opportunities.	Further investigation required
Low-Level Manipulation	The smart contracts do not use low-level calls and inline assembly.	Strong
Testing and Verification	Although the test suite is adequate, we uncovered issues that could have been caught only by a deeper test suite (e.g., TOB-SHER-4 , TOB-SHER-6).	Moderate

Summary of Findings

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

ID	Title	Type	Severity
1	Solidity compiler optimizations can be problematic	Undefined Behavior	Informational
2	Lack of events for critical operations	Auditing and Logging	Low
3	Infinite allowances pose security risks	Access Controls	Informational
4	claimReward function fails in paused system	Undefined Behavior	Low
5	TrueFi deposit can revert	Denial of Service	Medium
6	AlphaBetaEqualDepositMaxSplitter can start with a child whose value exceeds the maximum amount	Data Validation	Low
7	Lack of two-step process for contract ownership changes	Data Validation	High
8	USDC transfer could fail silently	Undefined Behavior	Low
9	Aave strategy could leave funds in the contract when withdrawing	Undefined Behavior	Low

Detailed Findings

1. Solidity compiler optimizations can be problematic

Severity: Informational

Difficulty: High

Type: Undefined Behavior

Finding ID: TOB-SHER-1

Target: `hardhat.config.js`

Description

The Sherlock contracts have enabled optional compiler optimizations in Solidity.

There have been several optimization bugs with security implications. Moreover, optimizations are **actively being developed**. Solidity compiler optimizations are disabled by default, and it is unclear how many contracts in the wild actually use them. Therefore, it is unclear how well they are being tested and exercised.

High-severity security issues due to optimization bugs **have occurred in the past**. A high-severity **bug in the emscripten-generated solc-js compiler** used by Truffle and Remix persisted until late 2018. The fix for this bug was not reported in the Solidity CHANGELOG. Another high-severity optimization bug resulting in incorrect bit shift results was **patched in Solidity 0.5.6**. More recently, another bug due to the **incorrect caching of keccak256** was reported.

A **compiler audit of Solidity** from November 2018 concluded that **the optional optimizations may not be safe**.

It is likely that there are latent bugs related to optimization and that new bugs will be introduced due to future optimizations.

Exploit Scenario

A latent or future bug in Solidity compiler optimizations—or in the Emscripten transpilation to `solc-js`—causes a vulnerability in the Sherlock contracts.

Recommendations

Short term, measure the gas savings from optimizations and carefully weigh them against the possibility of an optimization-related bug.

2. Lack of events for critical operations

Severity: Low

Difficulty: Low

Type: Auditing and Logging

Finding ID: TOB-SHER-2

Targets:

- contracts/managers/MasterStrategy.sol
- contracts/strategy/splitters/AlphaBetaSplitter.sol

Description

When depositing or withdrawing from a strategy, critical state changes in the contract do not emit events at the MasterStrategy level, the Splitter level, or the BaseStrategy level. The lack of events makes it difficult for users and blockchain-monitoring systems to review contracts for suspicious behavior after the contracts are deployed.

```
function deposit()
    external
    override(
        /*IStrategyManager, */
        INode
    )
    whenNotPaused
    onlySherlockCore
    balanceCache
{
    uint256 balance = want.balanceOf(address(this));
    if (balance == 0) revert InvalidConditions();

    want.safeTransfer(address(childOne), balance);

    childOne.deposit();
}
```

Figure 2.1: The `deposit()` function in `MasterStrategy.sol`#L117-L133

Exploit Scenario

An attacker, Eve, discovers a vulnerability in the MasterStrategy contract and modifies execution. Because this critical state change does not trigger the emission of events, this behavior can remain undetected until there has been significant damage to the system; this damage could include financial loss.

Recommendations

Short term, add events for any critical operations that produce state changes.

Long term, consider using a blockchain-monitoring system to track any suspicious behavior in the contracts. A monitoring mechanism for critical events would detect any compromised components.

3. Infinite allowances pose security risks

Severity: Informational

Difficulty: Low

Type: Access Controls

Finding ID: TOB-SHER-3

Target: contracts/strategy/*

Description

An infinite allowance of USDC is allotted for all five yield strategies (Compound, AAVE, Euler, Maple, and TrueFi). The following code for the CompoundStrategy illustrates this issue:

```
constructor(IMaster _initialParent) BaseNode(_initialParent) {  
    // Approve max USDC to cUSDC  
    want.safeIncreaseAllowance(address(CUSDC), type(uint256).max);  
}
```

Figure 3.1: The constructor() in *CompoundStrategy.sol*#L41-L44

However, if one of the strategy's protocols has a vulnerability that allows the transfer of tokens from an arbitrary address, this puts all funds currently in the strategy's contract at risk.

Exploit Scenario

An attacker, Eve, exploits one of Sherlock's protocols that allows arbitrary token transfers. Because all strategies have an infinite allowance, Eve can completely drain the compromised strategy's USDC balance.

Recommendations

Short term, give only as much allowance as is necessary for deposits and withdrawals.

Long term, minimize the possibilities of security issues due to a potential vulnerability in a third party protocol.

4. claimReward function fails in paused system

Severity: Low

Difficulty: Low

Type: Undefined Behavior

Finding ID: TOB-SHER-4

Target: contracts/strategy/*

Description

The whenNotPaused modifier is applied to the claimReward function, which prevents contract owners from claiming rewards when the system is in a paused state.

```
function claimReward() external whenNotPaused {
```

Figure 4.1: The claimReward() function in CompoundStrategy.sol#L102

This does not follow the documentation, which states the following: *Strategies are pausable, only depositing into the yield protocol will be paused.*

Exploit Scenario

Bob, the owner of a Sherlock contract, decides to pause the Compound strategy in order to disallow deposits and expects that doing so will allow him to claim the reward. However, the whenNotPaused modifier causes the call to the claimReward function to fail..

Recommendations

Short term, remove the whenNotPaused modifier from the claimReward function.

Long term, improve unit test coverage to uncover potential edge cases and ensure intended behavior throughout the system.

5. TrueFi deposit can revert

Severity: Medium

Difficulty: High

Type: Denial of Service

Finding ID: TOB-SHER-5

Target: contracts/strategy/TrueFiStrategy.sol

Description

In order to use the TrueFi strategy, a user must join the TrueFiPool to get tfUSDC by calling `tfUSDC.join`. Next, the user can call `tfFarm.stake` in order to stake their tfUSDC tokens to earn TRU tokens as a reward. However, the `hasShares` modifier can cause the `tfFarm.stake` call to revert. If the owner of `tfFarm` sets tfUSDC's rewards to 0, then it will be impossible for anyone to stake tfUSDC tokens, and calls to `Sherlock.yieldStrategyDeposit` can revert until the `TrueFiStrategy` is removed.

```
function _deposit() internal override whenNotPaused {
    //
    https://github.com/trusttoken/contracts-pre22/blob/main/contracts/truefi2/TrueFiPool
    2.sol#L469
    tfUSDC.join(want.balanceOf(address(this)));

    // How much tfUSDC did we receive because we joined the pool?
    uint256 tfUsdcBalance = tfUSDC.balanceOf(address(this));

    // Stake all tfUSDC in the tfFarm
    tfFarm.stake(tfUSDC, tfUsdcBalance);
}
```

Figure 5.1: The `_deposit()` function in `TrueFiStrategy.sol#L88-L97`

```
modifier hasShares(IERC20 token) {
    require(shares.staked[address(token)] > 0, "TrueMultiFarm: This token has no
    shares");
    _;
}
```

Figure 5.2: The `hasShares()` modifier in `TrueMultiFarm.sol#L101-L104`

Exploit Scenario

Bob, the owner of the Sherlock contract, wants to deposit USDC to start earning yield. Unbeknownst to him, however, the TrueFi farm's tfUSDC rewards were set to 0. This prevents him from calling `Sherlock.yieldStrategyDeposit`, which causes Sherlock stakers to lose yield.

Recommendations

Short term, check that `tfFarm.getShare` is greater than 0 before calling `tfFarm.stake`. Additionally, track whether or not there is USDC staked, since this will affect the `_balanceOf` and the `liquidExit` functions. When calling the `_balanceOf` function, the contract should use the `tfUSDC.balanceOf(address(this))` instead of the TrueMultiFarm staked amount. When using the `liquidExit` function, check if the tokens were staked before calling `tfFarm.unstake(tfUSDC, _amount);`.

Long term, check for functions' requirements or assumptions when integrating with a third-party protocol.

6. AlphaBetaEqualDepositMaxSplitter can start with a child whose value exceeds the maximum amount

Severity: Low

Difficulty: High

Type: Data Validation

Finding ID: TOB-SHER-6

Target: contracts/splitters/AlphaBetaEqualDepositMaxSplitter.sol

Description

The AlphaBetaEqualDepositMaxSplitter can start with a child contract whose value exceeds the maximum possible deposit amount.

This splitter can have one of the two child contracts with a maximum deposit amount; this amount is specified in the constructor. However, the splitter does not check that the child's value exceeds the limit, as shown below:

```
constructor(
    IMaster _initialParent,
    INode _initialChildOne,
    INode _initialChildTwo,
    uint256 _MIN_AMOUNT_FOR_EQUAL_SPLIT,
    uint256 _MAX_AMOUNT_FOR_CHILD_ONE,
    uint256 _MAX_AMOUNT_FOR_CHILD_TWO
)
    AlphaBetaEqualDepositSplitter(
        _initialParent,
        _initialChildOne,
        _initialChildTwo,
        _MIN_AMOUNT_FOR_EQUAL_SPLIT
    )
{
    // Either `_MAX_AMOUNT_FOR_CHILD_ONE` or `_MAX_AMOUNT_FOR_CHILD_TWO` has to be
    type(uint256).max
    if (_MAX_AMOUNT_FOR_CHILD_ONE != NO_LIMIT && _MAX_AMOUNT_FOR_CHILD_TWO !=
    NO_LIMIT) {
        revert InvalidArg();
    }

    // Either `_MAX_AMOUNT_FOR_CHILD_ONE` or `_MAX_AMOUNT_FOR_CHILD_TWO` has to be
    non type(uint256).max
    if (_MAX_AMOUNT_FOR_CHILD_ONE == NO_LIMIT && _MAX_AMOUNT_FOR_CHILD_TWO ==
    NO_LIMIT) {
        revert InvalidArg();
    }

    // Write variables to storage
```

```
MAX_AMOUNT_FOR_CHILD_ONE = _MAX_AMOUNT_FOR_CHILD_ONE;  
MAX_AMOUNT_FOR_CHILD_TWO = _MAX_AMOUNT_FOR_CHILD_TWO;  
}
```

*Figure 6.1: The constructor() function in
AlphaBetaEqualDepositMaxSplitter.sol#L40-L68*

Exploit Scenario

Bob, the owner of the Sherlock contract, wants to replace the current splitter with a new AlphaBetaEqualDepositMaxSplitter that will take over the replaced splitter's child contracts. However, the splitter will be deployed with a child whose balance exceeds the maximum possible amount.

Recommendations

Short term, check in the constructor of AlphaBetaEqualDepositMaxSplitter that the child with a maximum amount of deposited funds has a balance less than the maximum amount.

Long term, improve unit test coverage to uncover potential edge cases and ensure intended behavior throughout the system.

7. Lack of two-step process for contract ownership changes

Severity: High

Difficulty: High

Type: Data Validation

Finding ID: TOB-SHER-7

Target: contracts/strategy/base/BaseNode.sol

Description

The BaseNode contract inherits OpenZeppelin's Ownable contract, which provides a basic access control mechanism for the contract. However, the Ownable contract internally calls the `_transferOwnership()` function, which immediately sets the new owner of the contract. Making a critical change in a single step can introduce errors and lead to irrevocable mistakes.

```
function _transferOwnership(address newOwner) internal virtual {  
    address oldOwner = _owner;  
    _owner = newOwner;  
    emit OwnershipTransferred(oldOwner, newOwner);  
}
```

Figure 7.1: The `_transferOwnership()` function in `Ownable.sol#L71-L75`

Exploit Scenario

Bob, the owner of the Sherlock BaseNode contract, calls `transferOwnership()` but accidentally enters the wrong address. As a result, he permanently loses access to the contract.

Recommendations

Short term, perform ownership transfers through a two-step process in which the owner proposes a new address and the transfer is completed once the new address has executed a call to accept the role.

Long term, identify and document all possible actions that can be taken by privileged accounts and their associated risks. This will facilitate reviews of the codebase and prevent future mistakes.

8. USDC transfer could fail silently

Severity: Low

Difficulty: Medium

Type: Undefined Behavior

Finding ID: TOB-SHER-8

Target: contracts/strategy/splitters/AlphaBetaSplitter.sol

Description

The AlphaBetaSplitter contract allows deposits into each of its respective children by transferring USDC to the child contract and then calling `childOne.deposit()`. However, the transfer of USDC is performed by calling `transfer()` rather than using `safeTransfer()` and OpenZeppelin's SafeERC20 library. Since USDC is an upgradeable contract, the `want.transfer()` call may be changed to return a boolean. Since the return values are not checked, this transfer can lead to undefined behavior.

```
function _childOneDeposit(uint256 _amount) internal virtual {  
    // Transfer USDC to childOne  
    want.transfer(address(childOne), _amount);  
  
    // Signal childOne it received a deposit  
    childOne.deposit();  
}
```

Figure 8.1: The `_childOneDeposit()` function in `AlphaBetaSplitter.sol#L48-L54`

Exploit Scenario

The USDC contract is updated and the behavior of the transfer function is changed to return false rather than revert on failure. Bob, the owner of the Sherlock contracts, tries to make a deposit into a strategy. Because the USDC contract silently fails instead of reverting, Bob thinks he is earning yield; however, the child contract actually has no USDC to deposit.

Recommendations

Short term, use OpenZeppelin's SafeERC20 library for transfers of USDC. This will ensure that state changes are reverted if a transfer fails.

Long term, improve unit test coverage to uncover potential edge cases and ensure intended behavior throughout the system.

9. Aave strategy could leave funds in the contract when withdrawing

Severity: Low

Difficulty: Medium

Type: Undefined Behavior

Finding ID: TOB-SHER-9

Target: contracts/strategy/AaveStrategy.sol

Description

The Aave strategy could leave funds in the contract when withdrawing using `_withdrawAll`. The current implementation withdraws tokens from Aave to the Sherlock core contract; however, if USDC tokens are accidentally sent directly to the contract, the tokens will remain stuck. Additionally, note that all other strategies already check for this issue, and the contract's tokens will be sent to the Sherlock core contract.

```
function _withdrawAll() internal override returns (uint256) {
    ILendingPool lp = getLp();
    if (_balanceOf() == 0) {
        return 0;
    }
    // Withdraws all USDC from Aave's lending pool and sends it to core
    return lp.withdraw(address(want), type(uint256).max, core);
}
```

Figure 9.1: The `_withdrawAll()` function in `AaveStrategy.sol`#L96-L103

Exploit Scenario

Bob, a user, accidentally sends the tokens directly to the Aave strategy instead of depositing them. As a result, the tokens are lost.

Recommendations

Short term, check if the current contract holds tokens after the `AaveStrategy` calls to the `_withdrawAll` function; if the contract does hold tokens, send them to the Sherlock core contract.

Long term, try to minimize the possible consequences made by users' mistakes and keep the same expected behavior for related functions in different contracts.

Summary of Recommendations

The Sherlock contracts are a work in progress with multiple planned iterations. Trail of Bits recommends that Sherlock address the findings detailed in this report and take the following additional steps prior to deployment:

- Update the developer documentation to include the expected behavior when changing a splitter, and update the user documentation to explain the new yield strategies.
- Expand the test suite to cover cases that occur when the system is paused (TOB-SHER-4) or to simulate the possible states in which a dependent protocol can arrive (TOB-SHER-5).
- Develop a detailed incident response plan to ensure that any new issues can be addressed promptly and without confusion. For example, define the response if a withdrawal fails due to the underlying protocol's missing liquidity.

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
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. Code Quality Recommendations

This appendix lists code quality findings that we identified throughout our review.

- **Correct the following incorrect comments.** This will reduce confusion when reviewing the codebase.

```
// Signal childOne childTwo it received a deposit  
childTwo.deposit();
```

Figure C.1: *contracts/strategy/splitters/AlphaBetaSplitter.sol#L62-L63*

```
// If cUSDC.balanceOf(this) != 0, we can start to withdraw the cUSDC cUSDC  
if (cUSDCAmount != 0) {  
    // Revert if redeem function returns error code  
    if (CUSDC.redeem(cUSDCAmount) != 0) revert InvalidState();  
}
```

Figure C.2: *contracts/strategy/CompoundStrategy.sol#L75-L79*

- **Replace variable == false with !variable.** This will improve code readability.
- **Remove duplicate imports of the same file.**

```
import '../../../../interfaces/strategy/INode.sol';  
import '../../../../interfaces/strategy/INode.sol';
```

Figure C.3: *contracts/strategy/Base/BaseMaster.sol#L11-L12*