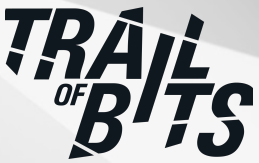

Sherlock Protocol

Security Assessment

June 3, 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

Test Coverage Disclaimer

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

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

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

Table of Contents

Executive Summary

Engagement Overview

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 static testing of the target system and its codebase, using both automated and manual processes..

Summary of Findings

The audit uncovered any significant flaws or defects that could impact system confidentiality, integrity, or availability. A summary of the findings and details on notable findings are provided below.

EXPOSURE ANALYSIS

Severity	*Count*
High	1
Medium	1
Low	5
Informational	2

CATEGORY BREAKDOWN

Category	*Count*
Access Controls	1
Auditing and Logging	1
Undefined Behavior	4
Data Validation	2
Denial of Service	1

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 is risky. 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

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 it possible that the depositing/withdrawing of funds into/from strategies fails when it shouldn't?
- Can the strategies lose funds?
- Do the tree strategies' actions have the correct checks?
- Is it possible to reach an incorrect state for the tree strategy?
- Is the `whenPaused` modifier applied correctly?
- Are there error-prone ownership transfers?
- 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 or the `sherlockCore` contract. Additionally we reviewed the safe transfer of contract's ownership, which led us to TOB-SHER-7 .
- We reviewed the individual strategies for possible errors when interacting with the corresponding protocol such as missing checks that could lead to reverting an operation when it shouldn't (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/adding a splitter/strategy or removing a strategy. This led us to the discovery of TOB-SHER-6.
- Since the strategies can be paused we reviewed the correct usage of the `whenNotPaused` modifier. This led us to TOB-SHER-4 where it was incorrectly applied to the `claimReward` function.
- We reviewed the deposit/withdrawal process from the splitter contract to its respective child contract, which led us to discover 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 `solc` 0.8.10 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. There is a single privileged actor who is able to do this.	**Satisfactory**
Complexity Management	The functionalities of each of the contracts 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 only be called by the owner of the Sherlock contract. Currently the owner is a multi-signature wallet controlled by the Sherlock team, however there are plans to transfer ownership to a DAO that 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,	**Moderate**

	and the documentation regarding the replacing of the splitters could be improved. Moreover, the user documentation has to 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	The test suite is adequate. However we uncovered issues that could have been caught with 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	Allocating infinite allowances poses security risks	Access Controls	**Informational**
4	ClaimReward function subject to paused system	Undefined Behavior	**Low**
5	TrueFi deposit can revert	Denial of Service	**Medium**
6	AlphaBetaEqualDepositMaxSplitter can start with a child already over the maximum amount	Data Validation	**Low**
7	Lack of two step process for contract ownership changes	Data Validation	**High**
8	USDC transfer can 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 security 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
Target: `contracts/managers/MasterStrategy.sol`,`contracts/strategy/splitters/AlphaBetaSplitter.sol`	

Description

When depositing or withdrawing from a strategy, events are not emitted neither at the MasterStrategy level, the Splitter level, nor the BaseStrategy level. As a result, it will be difficult to view the correct behavior of contracts once they are deployed. Without events, users and blockchain-monitoring systems cannot easily detect suspicious behavior.

```
  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 is able to modify execution. Without any events being generated, this behavior can go undetected until there has been significant damage to the system potentially including financial loss.

Recommendations

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

Long term, consider using a blockchain-monitoring system to monitor any suspicious events. This system would help quickly detect any compromised components.

3. Allocating infinite allowances poses security risks

Severity: **Informational**	Difficulty: **Low**
Type: Access Controls	Finding ID: TOB-SHER-3
Target: `contracts/strategy/*`	

Description
For all of the five yield strategies (Compound, AAVE, Euler, Maple, and TrueFi), an infinite allowance of USDC is allotted. As an illustration, consider the following code for the `CompoundStrategy`:

```
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 suffers from 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
One of the protocols used by Sherlock is exploited by an attacker, Eve. The exploit involves arbitrary token transfers. Because all of the strategies have an infinite allowance, Eve can completely drain the compromised strategy's USDC.

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

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

4. claimReward function subject to 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 however the documentation says the following: *Strategies are pausable, only depositing into the yield protocol will be paused.*

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

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

As a consequence, it won't be possible to call the `claimReward` function if the system is in a paused state which doesn't match the documentation.

Exploit Scenario
Bob, a Sherlock's owner, decides to pause the Compound strategy to not allow deposits expecting to be able to claim the reward however due to the whenNotPaused modifier the call will 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, one must join the `TrueFiPool` to get `tfUSDC` by calling `tfUSDC.join`. After, one can call `tfFarm.stake` in order to stake their `tfUSDC` tokens to earn TRU tokens as a reward. However, it's possible that the `tfFarm.stake` call will revert due to the `hasShares` modifier. If the `tfFarm`'s owner sets tfUSDC's rewards to 0, then it will be impossible for anyone to stake `tfUSDC` tokens, and possibly calling `Sherlock.yieldStrategyDeposit` will 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` and thus leads to a loss of yield for Sherlock stakers.

Recommendations

Short term, check that `tfFarm.getShare` is greater than 0 before calling `tfFarm.stake`. Additionally keep track if you staked or not since this will impact the `_balanceOf` and the `liquidExit` functions. The former in case of no staking should use the `tfUSDC.balanceOf(address(this))` instead of the TrueMultiFarm staked amount. The latter should also check if the tokens were staked before `tfFarm.unstake(tfUSDC, _amount);`.

Long term, when integrating with third party protocol make sure to check for functions' requirements or assumptions.

6. AlphaBetaEqualDepositMaxSplitter can start with a child already over 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 already over the maximum amount deposited possible.

This splitter can have one of the two childs with a maximum amount deposited which is specified in the constructor. However it doesn't check that the child with possibly a limit isn't already over it.

```
 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 a splitter with a new `AlphaBetaEqualDepositMaxSplitter` which will take the replaced splitter's childs. However it will be deployed with a child already having its balance greater than the maximum possible.

Recommendations

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

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 from 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 is error-prone and can 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 for a deposit into each of its respective children. It does so by transferring USDC to the child contract and calling `childOne.deposit()`. However, the transfer of USDC is done 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. However, 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 make sure that state changes are reverted in case of a failed transfer.

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 with `_withdrawAll`. The current implementation withdraws token from Aave to Sherlock core contract, however if tokens were sent by accident directly to the contract they will remain stuck. Additionally note that all the other strategies already check for that, 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, the tokens will be lost.

Recommendations

Short term, in the AaveStrategy's `_withdrawAll` function check if the current contract holds tokens, if yes 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:

- Improve the developer documentation to include the expected behavior when changing a splitter, and the user documentation to explain the new yield strategies.
- Expand the test suite to cover cases such as when the system is paused (TOB-SHER-4) or to simulate a particular state in which a dependent protocol can be (TOB-SHER-5).
- Develop a detailed incident response plan to ensure that any issues that arise can be addressed promptly and without confusion. For example what would you do in case 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 comments.** Having incorrect comments may lead to 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