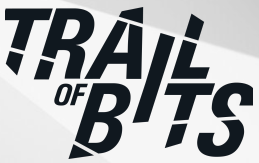

Sherlock Protocol

Retest Report

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

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.

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

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 scope, timeline, test targets, and coverage are provided in the original audit report.

Sherlock contracted Trail of Bits to review the fixes implemented for issues identified in the original report. On June 8, 2022, a team of two consultants conducted a review of the client-provided source code.

Summary of Findings

The original audit uncovered a significant flaw that could impact system confidentiality, integrity, or availability. A summary of the original findings is 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
Data Validation	2
Denial of Service	1
Undefined Behavior	4

Overview of Retest Results

Sherlock has sufficiently addressed a few of the issues described in the original audit report.

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
June 29, 2022	Delivery of retest report

Project Methodology

Our work in the retesting project included the following:

- A review of the findings in the original audit report
- A manual review of the client-provided source code and fix-related documentation

Project Targets

The engagement involved retesting of the <targets listed below OR 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/*`

Summary of Retest Results

The table below summarizes each of the original findings and indicates whether the issue has been sufficiently resolved.

ID	Title	Status
1	Solidity compiler optimizations can be problematic	Unresolved
2	Lack of events for critical operations	Unresolved
3	Infinite allowances pose security risks	Unresolved
4	claimReward function fails in paused system	Resolved
5	TrueFi deposit can revert	Resolved
6	AlphaBetaEqualDepositMaxSplitter can start with a child whose value exceeds the maximum amount	Unresolved
7	Lack of two-step process for contract ownership changes	Unresolved
8	USDC transfer could fail silently	Resolved
9	Aave strategy could leave funds in the contract when withdrawing	Unresolved

Detailed Retest Results

1. Solidity compiler optimizations can be problematic	
Status: **Unresolved**	
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.

Fix Analysis
The Sherlock team acknowledged the issue and decided to keep compiler optimizations on.

2. Lack of events for critical operations

Status: **Unresolved**	
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`

Fix Analysis

The Sherlock team chose not to address this issue because doing so would introduce extra calls to obtain the correct data. In some cases, this would increase gas costs and complexity.

3. Infinite allowances pose security risks	
Status: **Unresolved**	
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.

Fix Analysis

The Sherlock team chose not to address this issue since most of the time, all USDC in a strategy contract is supposed to be in the yield protocol.

4. claimReward function fails in paused system

Status: **Resolved**	
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()` 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.*

Fix Analysis

The Sherlock team fixed this issue by removing the `whenNotPaused` modifier from the `claimReward` function in the different strategies.

5. TrueFi deposit can revert

Status: **Resolved**	
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`

Fix Analysis

The Sherlock team fixed this issue. The `_deposit` function now checks if the shares allocated to `tfUSDC` in the `TrueMultiFarm` are greater than 0 before staking. The `_balanceOf` function now considers both the `tfUSDC` in the strategy contract and the

`tfUSDC` staked in `TrueMultiFarm`. The `liquidExit` function checks if the amount requested to withdraw is greater than the `tfUSDC` in the strategy contract plus the amount staked in `TrueMultiFarm`; if it is not and the amount to withdraw is greater than the balance in the strategy contract, it will proceed to unstake the difference from `TrueMultiFarm`.

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

Status: **Unresolved**	
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

Fix Analysis

The Sherlock team chose not to address this issue since adding this check would increase complexity; it is an edge case that occurs only if the owner does not catch it, and the protocol does not lose funds if it does occur.

7. Lack of two-step process for contract ownership changes	
Status: **Unresolved**	
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 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`

Fix Analysis

The Sherlock team chose not to address this issue because doing so would unnecessarily increase complexity; additionally, the team is confident that peer reviews of ownership changes will mitigate this issue.

8. USDC transfer could fail silently

Status: **Resolved**	
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`

Fix Analysis

The Sherlock team fixed this issue by using the SafeERC20 library for transfer of USDC.

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

Status: **Unresolved**	
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 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 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

Fix Analysis

The Sherlock team chose not to address this issue.

A. Status Categories

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

Retest 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.

B. 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.