Advanced Blockchain

Security Assessment

March 1, 2022

Prepared for:

Advanced Blockchain

Prepared by: **Nat Chin, Troy Sargent, 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 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

Table of Contents

Executive Summary

Engagement Overview

Advanced Blockchain engaged Trail of Bits to review the security of its Composable Finance vault and holding contracts. From February 7 to February 24, 2022, a team of two consultants conducted a security review of the client-provided source code, with six 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 the compromise of a smart contract, a loss of funds, or unexpected behavior. We conducted this audit with partial knowledge of the target system, including access to the source code and limited documentation. We performed static analysis and a manual review of the project's Solidity components.

Summary of Findings

The audit uncovered significant flaws that could result in the compromise of a smart contract, a loss of funds, or unexpected behavior. A summary of the findings and details on notable findings are provided below.

EXPOSURE ANALYSIS

Severity	*Count*
High	9
Medium	6
Low	7
Informational	5
Undetermined	2

CATEGORY BREAKDOWN

Category	*Count*
Data Validation	12
Undefined Behavior	5
Access Controls	3
Denial of Service	2
Configuration	2
Patching	2
Timing	2
Auditing and Logging	1

Notable Findings

Significant flaws that could impact the state of the funds held in a contract or cause unexpected behavior are listed below.

- **Excessive ownership privileges throughout the contracts (TOB-CMP-20)**

  Any function that can be invoked by a relayer can also be invoked by a contract owner. In all transactions, owners have complete discretion over the account to which funds are sent and the chain on which funds are sent. Although liquidity providers receive token shares as rewards, the amount they receive when redeeming those tokens is determined by arithmetic performed off-chain.

- **Widespread lack of data validation (TOB-CMP-16, TOB-CMP-17)**

  Certain data returned in function calls to untrusted external smart contracts is not validated. This lack of validation creates denial-of-service (DoS) attack vectors and can result in unsafe typecasting, which may lead to unexpected behavior.

- **Overly complicated contract and system architecture (TOB-CMP-17, TOB-CMP-21, TOB-CMP-22)**

  The system relies heavily on an off-chain system to manage the most critical operations, while the smart contracts primarily emit events to which the off-chain system responds. The primary concern with this design choice is the lack of on-chain safeguards to limit the abilities of the relayer. Secondarily, the intricate smart contract code makes it difficult to reason about the separation of privileges and abilities.

  The bridge architecture is also concerning. Given that L2 solutions like Arbitrum and Optimism have trustless bridges, it is not immediately clear why users would choose an architecture that requires a trusted actor.

  System and bridge architecture recommendations are provided in Appendix D and Appendix E, respectively.

- **Insufficient documentation (TOB-CMP-19)**

  The documentation is insufficient for such a complex system. In particular, the documentation lacks information on the overarching system assumptions and invariants, how the various protocol integrations work, how users recover their funds and accrued interest, and how the rebalancing bot works. Treating these aspects of the system as black boxes limited our ability to review the system as a whole.

Opportunities for Centralized Failures

A significant portion of the Composable Finance system relies on an off-chain relayer that must correctly perform actions on behalf of users. Because the relayer has the ability to unilaterally subvert the intentions of a smart contract, it is perhaps the most critical element of the system.

This section lays out several failure cases made possible by the smart contracts in their current state. It is included because the relayer component is outside the scope of this assessment and (to our knowledge) has not been reviewed by a third party.

The relayer bot is responsible for invoking functions on different blockchains after users have requested a transfer or withdrawal of funds. For the following reasons, this privileged role and architecture decision significantly increase the attack surface of the protocol and the likelihood of a system compromise:

- The correctness of the assumptions on the relayer bot's facilitation of cross-chain transactions has not been reviewed, and users cannot audit or verify the relayer code themselves.

- Composable Finance, through a single private key on each chain, has full discretion over user funds. Additionally, because the investment strategy data structure is mutable, users' funds may be deposited into arbitrary contracts.

- The relayer performs all operations in which funds are moved. There are no restrictions on the amount of funds that this externally owned account (EOA) can withdraw from the system.

- Users cannot exit the system and retrieve their funds without the relayer. Thus, if the relayer becomes nonfunctional or blocks users' actions, their funds will be stuck.

- Assets are not fungible across chains. This means that the risk profile of an asset depends on factors such as upgradeability and the security of each chain's validators. For example, an infinite mint of a token would render all tokens on one chain worthless and no longer directly redeemable on another chain's representation of the token. Thus, the movement of users' funds inherently exposes them to a loss of funds with no recourse.

- The relayer is also responsible for distributing interest earned by users who have deposited funds into the system. Because interest calculations are done off-chain, users cannot easily determine the amount of interest they will receive.

- The owner of a vault has control over the incentives provided to liquidity miners who loan funds through platforms such as Compound.

- The relayer sets the gas fees charged to users, which it pays on their behalf.

The Composable Finance system includes many privileged roles, including owner keys, bots, and the relayer. Appendix F provides a survey of those roles and the privileged operations they can perform to modify the contract system.

Cross-Blockchain Concerns

The interoperation of blockchains exposes the Composable Finance system to several risks; these include risks stemming from the idiosyncratic behavior of alternate Ethereum Virtual Machine (EVM) implementations, chain reorganizations (reorgs), and financial catastrophes like infinite mints.

Additionally, the interchain transfer and messaging implementation lacks cryptographic security and instead relies on strong trust assumptions. The critical nature of bridges demands a robust design and greater scrutiny. Furthermore, the bridges of L2 solutions like Optimism and Arbitrum are *trustless*; the solutions use asymmetric cryptography and fault proofs to secure user funds. These bridges provide levels of decentralization that are incomparable to that of the relayer bot.

Alternate EVM Implementations

The deployment of multiple blockchains requires the contracts to be deployed on each network. However, the contracts may execute differently on each one. The Arbitrum Virtual Machine, Optimism Virtual Machine, Avalanche C-Chain VM, and EVM running on mainnet all have unique implementations of the EVM.

All functionalities and opcodes in the contracts need to be carefully analyzed with regard to their execution on each blockchain. Time constraints prevented us from performing that analysis during the assessment.

Reorgs and the Limitations of Probabilistic Finality

The process of moving assets and data across chains relies on assumptions of partial synchrony and finality. In other words, it assumes that once an asset has been locked on one chain, the relayer can immediately release the asset to the user on another.

However, because the system relies on various consensus protocols, it is possible for a previous valid state to be invalidated by a new canonical chain that contains an entirely different state. This is known as a “reorg.” Consequently, there are weak guarantees surrounding the finality of asset transfers. If a transfer considered final by Composable Finance is plucked from the historical state, an attacker may be able to effectively double-spend tokens.

Conversely, a reorg could undo a transfer of funds from the relayer bot to a user; there is no documentation detailing how the user would reinitiate the transfer in that event.

It is necessary that Composable Finance run adequately peered nodes and wait an appropriate amount of time before considering a transfer confirmed. Composable Finance will need to investigate a solution for rebroadcasting and confirming transactions and to then develop and document it.

Financial Catastrophes

During the rebalancing process it performs on behalf of users, the Composable Finance system may experience a loss of funds. As assets change hands and move between chains, token contracts may be upgraded, experience bugs, or, in the case of asset-backed tokens, face insolvency. If a contract is upgraded or exploited, or if the amount of its collateral backing decreases, users’ funds may be lost or significantly decrease in value. This is an inherent risk of having custody of funds and integrating with a wide variety of blockchains and smart contracts.

To mitigate this risk, we suggest holding only non-upgradeable tokens that have been thoroughly reviewed and tested. It is important that stablecoins in particular have a track record of maintaining a healthy credit risk and that debt be removed from the system (liquidated) efficiently.

Project Summary

Contact Information

The following managers were associated with this project:

Dan Guido, Account Manager
dan@trailofbits.com

Mary O'Brien, Project Manager
mary.obrien@trailofbits.com

The following engineers were associated with this project:

Nat Chin, Consultant
natalie.chin@trailofbits.com

Troy Sargent, Consultant
troy.sargent@trailofbits.com

Anish Naik, Consultant
anish.naik@trailofbits.com

Project Timeline

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

Date	Event
February 3, 2022	Pre-project kickoff call
February 14, 2022	Status update meeting #1
February 22, 2022	Status update meeting #2
March 1, 2022	Delivery of final report
March 1, 2022	Report readout meeting

Project Goals

The engagement was scoped to provide a security assessment of the Composable Finance vault and holding contracts. Specifically, we sought to answer the following non-exhaustive list of questions:

- Could an attacker steal funds from the system?
- Are there appropriate access controls in place for user and admin operations?
- Could an attacker trap the system?
- Are there DoS attack vectors?
- Does the bridge track token balances correctly?
- Could users lose access to their funds?
- Could an attacker front-run a token swap?
- Are users given guarantees that they will indeed receive the funds they are owed?
- Are fee amounts validated and limited?
- Is data from external contracts validated?
- Do the smart contracts behave consistently on various L1 and L2 networks?

Project Targets

The engagement involved a review and testing of the targets listed below.

Crosslayer Portal

Repository	https://github.com/ComposableFi/CrosslayerPortal
Version	`4ffd6104b333f67ffe9ef1f4acd069c899ef030f`
Type	Solidity
Platforms	Ethereum / EVM chains

V2 SDK Contracts

Repository	https://github.com/ComposableFi/v2-sdk-protocols
Version	`1fbb3fbb64d796134b58c1c5f024e4185cd4bdee`
Type	Solidity
Platforms	Ethereum / EVM chains

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:

`MosaicVault`. This upgradeable contract contains the core logic for depositing liquidity into the Mosaic protocol and withdrawing liquidity from it. The contract controls the funds that are transferred across blockchains by sending them to the `MosaicHolding` contract. We focused on reviewing the `MosaicVault` contract's deposit, withdrawal, and transfer functionalities.

`MosaicHolding`. This upgradeable contract holds all funds deposited into the `MosaicVault` contract, invests funds in third-party protocols to secure additional annual percentage yield (APY) for users, and claims the interest-bearing tokens that investors earn on liquidity provided to third-party protocols. We focused on reviewing the functionalities involved in covering withdrawal requests, claiming interest-bearing tokens, and investing tokens.

`functionCalls`. This set of upgradeable smart contracts facilitates function calls across blockchains. We primarily reviewed the creation and handling of cross-chain message call requests.

NFT ERC721 contracts. This set of upgradeable contracts includes the `MosaicNFT` and `Summoner` contracts, which enable cross-chain ERC721 transfers. We focused on the contracts' locking and sealing functionalities and the system architecture.

Third-party protocol contracts. These upgradeable contracts enable integrations with third-party DeFi platforms such as Uniswap, Balancer, Aave, Compound, Sushiswap, and Polygon Reward. We checked whether the third-party integrations adhere to best practices.

V2 SDK contracts. These upgradeable contracts allow other developers to use the Crosslayer Portal code. We focused on the functions that interact directly with the `MosaicVault`, `Summoner`, and `MsgSender` contracts. We also reviewed the functions that interact with the Chainlink system.

Coverage Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. During this project, we were unable to perform comprehensive testing of the following system elements, which may warrant further review:

Mosaic Bridge. The bridge enables transfers of funds from L1 to L2 blockchains. Because the bridge is not intended to be part of the Phase 2 release, it was outside the scope of this audit.

Bribe Protocol. The Bribe Protocol is a governance protocol that attempts to incentivize decentralized autonomous organization (DAO) participation. Users deposit their governance tokens into a pool; they receive rewards when bidders pay to use the collective voting power of the pool to vote on an existing proposal. Because of the critical nature of the Composable Finance issues identified in the audit, this codebase was considered outside of its scope.

Off-chain components. The architecture is heavily reliant on off-chain components. All of the off-chain components were considered outside the scope of this audit.

Relayer bot assumptions. The most critical part of the system is the relayer bot, tasked with executing critical operations (e.g., withdrawals and transfers) by forwarding transactions on behalf of users. Because this component was out of scope, Trail of Bits made various assumptions on the validation performed by the relayer bot, including the following:

- **The bot accounts for differences in block timestamps across blockchains.** Because the system will be deployed across a variety of blockchains, the bot must account for different block confirmation times as well as block number differences caused by transfer delays and reward calculations. The relayer bot should ensure that users will not receive extra rewards for transactions on chains that progress faster than others.

- **Tokens are distributed fairly and in the correct amount.** The relayer is tasked with not only calculating the token amounts to be distributed but also distributing tokens when users withdraw their funds. Because user balances are not reflected on-chain, the relayer must correctly track the balances and credit *pro rata* interest earnings to users. In conducting the audit, we assumed the implementation of this logic to be correct.

- **The relayer bot safely executes the locks and seals on NFTs.** The NFT contracts rely on the relayer to lock and release the NFTs held by the Mosaic contracts. Thus, the relayer must use the correct network ID when an NFT is summoned and must never allow more than one version of the same NFT to exist in a released state.

- **The relayer adheres to the `transferDelay` variable provided by users to NFT contracts.** The `transferDelay` variable allows a user to specify the amount of time that a transfer should be considered valid. The relayer is trusted to use this value as intended.

- **The relayer uses a single queue of a finite size to store emitted events.** The relayer bot is responsible for acting upon events that are emitted when a user wishes to make a cross-layer message call. These events are assumed to be stored in a single queue of a finite size.

- **The relayer handles fund movements appropriately.** The relayer is in charge of moving funds between vaults on different blockchains and executing withdrawal requests. Because the relayer has the ability to call multiple functions to move funds, the relayer must behave non-maliciously.

- **The relayer keeps track of all failed transactions and fairly charges fees to users.** The relayer needs to keep track of all fees charged for user transactions. When a transaction is successful, the relayer deducts the user's fee from it; in case of a failure, the relayer must return the transaction fee to the user.

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	Solidity v0.8.0 arithmetic operations are used in the system but are not tested through automated testing or unit tests. The formulas in the codebase are not documented (TOB-CMP-19). The reward token bookkeeping and the undocumented reward calculations done off-chain by the relayer are error-prone (TOB-CMP-5).	Weak
Auditing	While some functions emit events for critical operations and effectively monitor on-chain activity, many functions that perform critical operations do not emit events (TOB-CMP-28). The documentation we were provided lacks information on the use of off-chain components in behavior monitoring as well as a formal incident response plan. Additionally, it is unclear how users can verify the integrity of the relayer's actions in executing cross-chain calls. The relayer's handling of edge cases such as duplicate messages and chain reorgs is similarly unclear.	Weak
Authentication / Access Controls	The access controls do not follow the principle of least privilege; instead, they vest too much power in one admin. Users cannot call the functions that facilitate exits from the system (TOB-CMP-27).	Weak
Complexity Management	The token integrations are nebulous and complex (TOB-CMP-21). The functions are verbose and have excessive branching (TOB-CMP-22).	Weak

Cryptography and Key Management	Contract owners and the relayer are controlled by single private keys. Key management is a critical point of failure in the system. Exposure of these private keys would allow an attacker to steal all funds from the protocol (TOB-CMP-26).	Weak
Decentralization	The private key that controls a contract's owner has complete control of user funds, and a user cannot exit the system unless this entity signs a transaction (Appendix F).	Weak
Documentation	We received limited documentation that lacks details on various process flows and were not provided with a clear specification (TOB-CMP-19).	Weak
Front-Running Resistance	The system is vulnerable to sandwich attacks (TOB-CMP-2) and initialization front-running (TOB-CMP-1).	Weak
Low-Level Calls	The system uses low-level calls that are not properly safeguarded, and gas-intensive operations, namely memory expansion operations, create DoS attack vectors in the system (TOB-CMP-3).	Weak
Testing and Verification	The system does not detail its security properties, and many assumptions are unstated. The test suite reflects the lack of a specification and should be expanded. Additionally, dependency issues prevented Trail of Bits from running the test suite (TOB-CMP-29).	Weak

Summary of Findings

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

ID	Title	Type	Severity
1	Initialization functions can be front-run	Configuration	High
2	Trades are vulnerable to sandwich attacks	Timing	High
3	forwardCall creates a denial-of-service attack vector	Denial of Service	High
8	Lack of two-step process for contract ownership changes	Data Validation	High
9	sendFunds is vulnerable to reentrancy by owners	Timing	High
20	MosaicVault and MosaicHolding owner has excessive privileges	Access Controls	High
24	SushiswapLiquidityProvider deposits cannot be used to cover withdrawal requests	Data Validation	High
26	MosaicVault and MosaicHolding owner is controlled by a single private key	Access Controls	High
27	The relayer is a single point of failure	Access Controls	High
4	Project dependencies contain vulnerabilities	Patching	Medium
10	DoS risk created by cross-chain message call requests on certain networks	Denial of Service	Medium
11	Missing validation in takeFees function	Data Validation	Medium

12	Unimplemented getAmountsOut function in Balancer V2	Undefined Behavior	**Medium**
28	Lack of events for critical operations	Auditing and Logging	**Medium**
30	Insufficient protection of sensitive information	Configuration	**Medium**
5	Accrued interest is not attributable to the underlying investor on-chain	Data Validation	**Low**
6	User funds can become trapped in nonstandard token contracts	Data Validation	**Low**
13	Use of MsgReceiver to check _feeToken status leads to unnecessary gas consumption	Data Validation	**Low**
14	Active liquidity providers can set arbitrary _tokenOut values when withdrawing liquidity	Data Validation	**Low**
15	Withdrawal assumptions may lead to transfers of an incorrect token	Data Validation	**Low**
16	Improper validation of Chainlink data	Data Validation	**Low**
25	Incorrect safeIncreaseAllowance() amount can cause invest() calls to revert	Data Validation	**Low**
17	Incorrect check of token status in the providePassiveLiquidity function	Data Validation	**Informational**
18	Solidity compiler optimizations can be problematic	Undefined Behavior	**Informational**
19	Lack of contract documentation	Undefined Behavior	**Informational**
21	Unnecessary complexity due to interactions with native and smart contract tokens	Data Validation	**Informational**

29	Use of legacy openssl version in CrosslayerPortal tests	Patching	**Informational**
22	Commented-out and unimplemented conditional statements	Undefined Behavior	**Undetermined**
23	Error-prone NFT management in the Summoner contract	Undefined Behavior	**Undetermined**

Detailed Findings

1. Initialization functions can be front-run

Severity: **High**	Difficulty: **High**
Type: Configuration	Finding ID: TOB-CMP-1
Target: Throughout the contracts	

Description

The `CrosslayerPortal` contracts have initializer functions that can be front-run, allowing an attacker to incorrectly initialize the contracts. Due to the use of the `delegatecall` proxy pattern, these contracts cannot be initialized with their own constructors, and they have initializer functions:

```
function initialize() public initializer {
    __Ownable_init();
    __Pausable_init();
    __ReentrancyGuard_init();
}
```

Figure 1.1: The `initialize` function in `MsgSender:126-130`

An attacker could front-run these functions and initialize the contracts with malicious values.

This issue affects the following system contracts:

- `contracts/core/BridgeAggregator`
- `contracts/core/InvestmentStrategyBase`
- `contracts/core/MosaicHolding`
- `contracts/core/MosaicVault`
- `contracts/core/MosaicVaultConfig`
- `contracts/core/functionCalls/MsgReceiverFactory`
- `contracts/core/functionCalls/MsgSender`
- `contracts/nfts/Summoner`
- `contracts/protocols/aave/AaveInvestmentStrategy`
- `contracts/protocols/balancer/BalancerV1Wrapper`
- `contracts/protocols/balancer/BalancerVaultV2Wrapper`
- `contracts/protocols/bancor/BancorWrapper`

- `contracts/protocols/compound/CompoundInvestmentStrategy`
- `contracts/protocols/curve/CurveWrapper`
- `contracts/protocols/gmx/GmxWrapper`
- `contracts/protocols/sushiswap/SushiswapLiquidityProvider`
- `contracts/protocols/synapse/ISynapseSwap`
- `contracts/protocols/synapse/SynapseWrapper`
- `contracts/protocols/uniswap/IUniswapV2Pair`
- `contracts/protocols/uniswap/UniswapV2Wrapper`
- `contracts/protocols/uniswap/UniswapWrapper`

Exploit Scenario

Bob deploys the `MsgSender` contract. Eve front-runs the contract's initialization and sets her own address as the owner address. As a result, she can use the `initialize` function to update the contract's variables, modifying the system parameters.

Recommendations

Short term, to prevent front-running of the initializer functions, use `hardhat-deploy` to initialize the contracts or replace the functions with constructors. Alternatively, create a deployment script that will emit sufficient errors when an `initialize` call fails.

Long term, carefully review the Solidity documentation, especially the "Warnings" section, as well as the pitfalls of using the `delegatecall` proxy pattern.

2. Trades are vulnerable to sandwich attacks	
Severity: **High**	Difficulty: **Medium**
Type: Timing	Finding ID: TOB-CMP-2
Target: `CrossLayerPortal/core/nativeSwappers/MosaicNativeSwapperETH.sol`, `MosaicNativeSwapperAVAX.sol`	

Description

The `swapToNative` function does not allow users to specify the `minAmountOut` parameter of `swapExactTokensForETH`, which indicates the minimum amount of ETH that a user will receive from a trade. Instead, the value is hard-coded to zero, meaning that there is no guarantee that users will receive any ETH in exchange for their tokens. By using a bot to sandwich a user's trade, an attacker could increase the slippage incurred by the user and profit off of the spread at the user's expense.

The `minAmountOut` parameter is meant to prevent the execution of trades through illiquid pools and to provide protection against sandwich attacks. The current implementation lacks protections against high slippage and may cause users to lose funds. This applies to the AVAX version, too. Composable Finance indicated that only the relayer will call this function, but the function lacks access controls to prevent users from calling it directly.

Importantly, it is highly likely that if a relayer does not implement proper protections, all of its trades will suffer from high slippage, as they will represent pure-profit opportunities for sandwich bots.

```
uint256[] memory amounts = swapRouter.swapExactTokensForETH(
    _amount,
    0,
    path,
    _to,
    deadline
);
```

Figure 2.1: Part of the SwapToNative function in `MosaicNativeSwapperETH.sol: 44–50`

Exploit Scenario

Bob, a relayer, makes a trade on behalf of a user. The `minAmountOut` value is set to zero, which means that the trade can be executed at any price. As a result, when Eve sandwiches the trade with a buy and sell order, Bob sells the tokens without purchasing any, effectively giving away tokens for free.

Recommendations

Short term, allow users (relayers) to input a slippage tolerance, and add access controls to the `swapToNative` function.

Long term, consider the risks of integrating with other protocols such as Uniswap and implement mitigations for those risks.

3. forwardCall creates a denial-of-service attack vector

Severity: **High**	Difficulty: **Medium**
Type: Denial of Service	Finding ID: TOB-CMP-3
Target: `CrossLayerPortal/contracts/functionCalls/MsgReceiver.sol`	

Description

Low-level external calls can exhaust all available gas by returning an excessive amount of data, thereby causing the relayer to incur memory expansion costs. This can be used to cause an out-of-gas exception and is a denial-of-service (DoS) attack vector. Since arbitrary contracts can be called, Composable Finance should implement additional safeguards.

If an out-of-gas exception occurs, the message will never be marked as forwarded (`forwarded[id] = true`). If the relayer repeatedly retries the transaction, assuming it will eventually be marked as forwarded, the queue of pending transactions will grow without bounds, with each unsuccessful message-forwarding attempt carrying a gas cost. The `approveERC20TokenAndForwardCall` function is also vulnerable to this DoS attack.

```
(success, returnData) = _contract.call{value: msg.value}(_data);
require(success, "Failed to forward function call");

uint256 balance = IERC20(_feeToken).balanceOf(address(this));
require(balance >= _feeAmount, "Not enough tokens for the fee");
forwarded[_id] = true;
```

Figure 3.1: Part of the `forwardCall` function in `MsgReceiver:79-85`

Exploit Scenario

Eve deploys a contract that returns 10 million bytes of data. A call to that contract causes an out-of-gas exception. Since the transaction is not marked as forwarded, the relayer continues to propagate the transaction without success. This results in excessive resource consumption and a degraded quality of service.

Recommendations

Short term, require that the size of return data be fixed to 32 bytes.

Long term, review the documentation on low-level Solidity calls and EVM edge cases.

References

- Excessively Safe Call

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

Severity: **High**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-CMP-8
Target: `@openzeppelin/contracts-upgradeable/access/OwnableUpgradeable.sol`	

Description

The owner of a contract in the Composable Finance ecosystem can be changed through a call to the `transferOwnership` function. This function internally calls the `setOwner` function, which immediately sets the contract's new owner. Making such a critical change in a single step is error-prone and can lead to irrevocable mistakes.

```
/**
 * @dev Leaves the contract without owner. It will not be possible to call
 * `onlyOwner` functions anymore. Can only be called by the current owner.
 *
 * NOTE: Renouncing ownership will leave the contract without an owner,
 * thereby removing any functionality that is only available to the owner.
 */
function renounceOwnership() public virtual onlyOwner {
    _setOwner(address(0));
}

/**
 * @dev Transfers ownership of the contract to a new account (`newOwner`).
 * Can only be called by the current owner.
 */
function transferOwnership(address newOwner) public virtual onlyOwner {
    require(newOwner != address(0), "Ownable: new owner is the zero address");
    _setOwner(newOwner);
}

function _setOwner(address newOwner) private {
    address oldOwner = _owner;
    _owner = newOwner;
    emit OwnershipTransferred(oldOwner, newOwner);
}
```

Figure 8.1: OpenZeppelin's `OwnableUpgradeable` contract

Exploit Scenario
Bob, a Composable Finance developer, invokes `transferOwnership()` to change the address of an existing contract's owner 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.

9. sendFunds is vulnerable to reentrancy by owners	
Severity: **High**	Difficulty: **Low**
Type: Timing	Finding ID: TOB-CMP-9
Target: `CrosslayerPortal/contracts/PolygonReward/FundKeeper.sol`	

Description

The `sendFunds` function is vulnerable to reentrancy and can be used by the owner of a token contract to drain the contract of its funds. Specifically, because `fundsTransfered[user]` is written to after a call to an external contract, the contract's owner could input his or her own address and reenter the `sendFunds` function to drain the contract's funds. An owner could send funds to him- or herself without using the reentrancy, but there is no reason to leave this vulnerability in the code.

Additionally, the `FundKeeper` contract can send funds to any user by calling `setAmountToSend` and then `sendFunds`. It is unclear why `amountToSend` is not changed (set to zero) after a successful transfer. It would make more sense to call `setAmountToSend` after each transfer and to store users' balances in a mapping.

```
function setAmountToSend(uint256 amount) external onlyOwner {
    amountToSend = amount;
    emit NewAmountToSend(amount);
}

function sendFunds(address user) external onlyOwner {
    require(!fundsTransfered[user], "reward already sent");
    require(address(this).balance >= amountToSend, "Contract balance low");

    // solhint-disable-next-line avoid-low-level-calls
    (bool sent, ) = user.call{value: amountToSend}("");
    require(sent, "Failed to send Polygon");

    fundsTransfered[user] = true;
    emit FundSent(amountToSend, user);
}
```

Figure 9.1: Part of the `sendFunds` function in `FundKeeper.sol:23-38`

Exploit Scenario

Eve's smart contract is the owner of the `FundKeeper` contract. Eve's contract executes a transfer for which Eve should receive only 1 ETH. Instead, because the user address is a contract with a fallback function, Eve can reenter the `sendFunds` function and drain all ETH from the contract.

Recommendations

Short term, set `fundsTransfered[user]` to `true` prior to making external calls.

Long term, store each user's balance in a mapping to ensure that users cannot make withdrawals that exceed their balances. Additionally, follow the checks-effects-interactions pattern.

20. MosaicVault and MosaicHolding owner has excessive privileges	
Severity: **High**	Difficulty: **High**
Type: Access Controls	Finding ID: TOB-CMP-20
Target: `CrosslayerPortal`	

Description

The owner of the `MosaicVault` and `MosaicHolding` contracts has too many privileges across the system. Compromise of the owner's private key would put the integrity of the underlying system at risk.

The owner of the `MosaicVault` and `MosaicHolding` contracts can perform the following privileged operations in the context of the contracts:

- Rescuing funds if the system is compromised
- Managing withdrawals, transfers, and fee payments
- Pausing and unpausing the contracts
- Rebalancing liquidity across chains
- Investing in one or more investment strategies
- Claiming rewards from one or more investment strategies

The ability to drain funds, manage liquidity, and claim rewards creates a single point of failure. It increases the likelihood that the contracts' owner will be targeted by an attacker and increases the incentives for the owner to act maliciously.

Exploit Scenario

Alice, the owner of `MosaicVault` and `MosaicHolding`, deploys the contracts. `MosaicHolding` eventually holds assets worth USD 20 million. Eve gains access to Alice's machine, upgrades the implementations, pauses `MosaicHolding`, and drains all funds from the contract.

Recommendations

Short term, clearly document the functions and implementations that the owner of the `MosaicVault` and `MosaicHolding` contracts can change. Additionally, split the privileges provided to the owner across multiple roles (e.g., a fund manager, fund rescuer, owner, etc.) to ensure that no one address has excessive control over the system.

Long term, develop user documentation on all risks associated with the system, including those associated with privileged users and the existence of a single point of failure.

24. SushiswapLiquidityProvider deposits cannot be used to cover withdrawal requests

Severity: **High**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-CMP-24
Target: `CrosslayerPortal/contracts/core/MosaicHolding.sol`, `CrosslayerPortal/contracts/protocols/sushiswap/SushiswapLiquidityProvider.sol`	

Description

Withdrawal requests that require the removal of liquidity from a Sushiswap liquidity pool will revert and cause a system failure.

When a user requests a withdrawal of liquidity from the Mosaic system, `MosaicVault` (via the `coverWithdrawRequest()` function) queries `MosaicHolding` to see whether liquidity must be removed from an investment strategy to cover the withdrawal amount (figure 24.1).

```
function _withdraw(
    address _accountTo,
    uint256 _amount,
    address _tokenIn,
    address _tokenOut,
    uint256 _amountOutMin,
    WithdrawData calldata _withdrawData,
    bytes calldata _data
)
    internal
    onlyWhitelistedToken(_tokenIn)
    validAddress(_tokenOut)
    nonReentrant
    onlyOwnerOrRelayer
    whenNotPaused
    returns (uint256 withdrawAmount)
{
    IMosaicHolding mosaicHolding = IMosaicHolding(vaultConfig.getMosaicHolding());
    require(hasBeenWithdrawn[_withdrawData.id] == false, "ERR: ALREADY WITHDRAWN");
    if (_tokenOut == _tokenIn) {
        require(
            mosaicHolding.getTokenLiquidity(_tokenIn,
_withdrawData.investmentStrategies) >=
                _amount,
            "ERR: VAULT BAL"
        );
    }
[...]

    mosaicHolding.coverWithdrawRequest(
        _withdrawData.investmentStrategies,
        _tokenIn,
        withdrawAmount
    );
[...]
}
```

Figure 24.1: The `_withdraw` function in `MosaicVault:404-474`

If `MosaicHolding`'s balance of the token being withdrawn (`_tokenIn`) is not sufficient to cover the withdrawal, `MosaicHolding` will iterate through each investment strategy in the `_investmentStrategy` array and remove enough `_tokenIn` to cover it. To remove liquidity from an investment strategy, it calls `withdrawInvestment()` on that strategy (figure 24.2).

```
function coverWithdrawRequest(
    address[] calldata _investmentStrategies,
    address _token,
    uint256 _amount
) external override {
    require(hasRole(MOSAIC_VAULT, msg.sender), "ERR: PERMISSIONS A-V");
    uint256 balance = IERC20(_token).balanceOf(address(this));
    if (balance >= _amount) return;

    uint256 requiredAmount = _amount - balance;
    uint8 index;
    while (requiredAmount > 0) {
        address strategy = _investmentStrategies[index];
        IInvestmentStrategy investment = IInvestmentStrategy(strategy);
        uint256 investmentAmount = investment.investmentAmount(_token);
        uint256 amountToWithdraw = 0;
        if (investmentAmount >= requiredAmount) {
            amountToWithdraw = requiredAmount;
            requiredAmount = 0;
        } else {
            amountToWithdraw = investmentAmount;
            requiredAmount = requiredAmount - investmentAmount;
        }
        IInvestmentStrategy.Investment[]
            memory investments = new IInvestmentStrategy.Investment[](1);
        investments[0] = IInvestmentStrategy.Investment(_token, amountToWithdraw);
        IInvestmentStrategy(investment).withdrawInvestment(investments, "");

        emit InvestmentWithdrawn(strategy, msg.sender);

        index++;
    }
    require(IERC20(_token).balanceOf(address(this)) >= _amount, "ERR: VAULT BAL");
}
```

Figure 24.2: The `coverWithdrawRequest` function in `MosaicHolding:217-251`

This process works for an investment strategy in which the `investments` array function argument has a length of 1. However, in the case of `SushiswapLiquidityProvider`, the `withdrawInvestment()` function expects the `investments` array to have a length of 2 (figure 24.3).

```
function withdrawInvestment(Investment[] calldata _investments, bytes calldata
_data)
    external
    override
    onlyInvestor
    nonReentrant
{
    Investment memory investmentA = _investments[0];
    Investment memory investmentB = _investments[1];
    (uint256 deadline, uint256 liquidity) = abi.decode(_data, (uint256, uint256));
    IERC20Upgradeable pair = IERC20Upgradeable(getPair(investmentA.token,
investmentB.token));
    pair.safeIncreaseAllowance(address(sushiSwapRouter), liquidity);
    (uint256 amountA, uint256 amountB) = sushiSwapRouter.removeLiquidity(
        investmentA.token,
        investmentB.token,
        liquidity,
        investmentA.amount,
        investmentB.amount,
        address(this),
        deadline
    );

    IERC20Upgradeable(investmentA.token).safeTransfer(mosaicHolding, amountA);
    IERC20Upgradeable(investmentB.token).safeTransfer(mosaicHolding, amountB);
}
```

Figure 24.3: The `withdrawInvestment` function in `SushiswapLiquidityProvider:90-113`

Thus, any withdrawal request that requires the removal of liquidity from `SushiswapLiquidityProvider` will revert.

Exploit Scenario

Alice wishes to withdraw liquidity (`tokenA`) that she deposited into the Mosaic system. The `MosaicHolding` contract does not hold enough `tokenA` to cover the withdrawal and thus tries to withdraw `tokenA` from the `SushiswapLiquidityProvider` investment strategy. The request reverts, and Alice's withdrawal request fails, leaving her unable to access her funds.

Recommendations

Short term, avoid depositing user liquidity into the `SushiswapLiquidityProvider` investment strategy.

Long term, take the following steps:

- Identify and implement one or more data structures that will reduce the technical debt resulting from the use of the `InvestmentStrategy` interface.
- Develop a more effective solution for covering withdrawals that does not consistently require withdrawing funds from other investment strategies.

26. MosaicVault and MosaicHolding owner is controlled by a single private key

Severity: **High**	Difficulty: **High**
Type: Access Controls	Finding ID: TOB-CMP-26
Target: `CrosslayerPortal`	

Description

The `MosaicVault` and `MosaicHolding` contracts manage many critical functionalities, such as those for rescuing funds, managing liquidity, and claiming rewards.

The owner of these contracts is a single externally owned account (EOA). As mentioned in TOB-CMP-20, this creates a single point of failure. Moreover, it makes the owner a high-value target for attackers and increases the incentives for the owner to act maliciously. If the private key is compromised, the system will be compromised too.

Exploit Scenario

Alice, the owner of the `MosaicVault` and `MosaicHolding` contracts, deploys the contracts. `MosaicHolding` eventually holds assets worth USD 20 million. Eve gains access to Alice's machine, upgrades the implementations, pauses `MosaicHolding`, and drains all funds from the contract.

Recommendations

Short term, change the owner of the contracts from a single EOA to a multi-signature account.

Long term, take the following steps:

- Develop user documentation on all risks associated with the system, including those associated with privileged users and the existence of a single point of failure.
- Assess the system's key management infrastructure and document the associated risks as well as an incident response plan.

27. The relayer is a single point of failure	
Severity: **High**	Difficulty: **High**
Type: Access Controls	Finding ID: TOB-CMP-27
Target: `CrosslayerPortal`	

Description
Because the relayer is a centralized service that is responsible for critical functionalities, it constitutes a single point of failure within the Mosaic ecosystem.

The relayer is responsible for the following tasks:

- Managing withdrawals across chains
- Managing transfers across chains
- Managing the accrued interest on all users' investments
- Executing cross-chain message call requests
- Collecting fees for all withdrawals, transfers, and cross-chain message calls
- Refunding fees in case of failed transfers or withdrawals

The centralized design and importance of the relayer increase the likelihood that the relayer will be targeted by an attacker.

Exploit Scenario
Eve, an attacker, is able to gain root access on the server that runs the relayer. Eve can then shut down the Mosaic system by stopping the relayer service. Eve can also change the source code to trigger behavior that can lead to the drainage of funds.

Recommendations
Short term, document an incident response plan and monitor exposed ports and services that may be vulnerable to exploitation.

Long term, arrange an external security audit of the core and peripheral relayer source code. Additionally, consider implementing a decentralized relayer architecture more resistant to system takeovers.

4. Project dependencies contain vulnerabilities	
Severity: **Medium**	Difficulty: **Low**
Type: Patching	Finding ID: TOB-CMP-4
Target: `CrosslayerPortal`	

Description

Although dependency scans did not yield a direct threat to the project under review, `npm` and `yarn audit` identified dependencies with known vulnerabilities. Due to the sensitivity of the deployment code and its environment, it is important to ensure dependencies are not malicious. Problems with dependencies in the JavaScript community could have a significant effect on the repositories under review. The output below details these issues.

CVE ID	Description	Dependency
CVE-2021-23337	Command Injection in lodash	`lodash`
CVE-2022-0235	node-fetch is vulnerable to Exposure of Sensitive Information to an Unauthorized Actor	`node-fetch`
CVE-2021-23358	Arbitrary Code Execution in underscore	`underscore`

Figure 4.1: Advisories affecting `CrosslayerPortal` dependencies

Exploit Scenario

Alice installs the dependencies of an in-scope repository on a clean machine. Unbeknownst to Alice, a dependency of the project has become malicious or exploitable. Alice subsequently uses the dependency, disclosing sensitive information to an unknown actor.

Recommendations

Short term, ensure dependencies are up to date. Several node modules have been documented as malicious because they execute malicious code when installing dependencies to projects. Keep modules current and verify their integrity after installation.

Long term, consider integrating automated dependency auditing into the development workflow. If a dependency cannot be updated when a vulnerability is disclosed, ensure the code does not use and is not affected by the vulnerable functionality of the dependency.

10. DoS risk created by cross-chain message call requests on certain networks	
Severity: **Medium**	Difficulty: **Medium**
Type: Denial of Service	Finding ID: TOB-CMP-10
Target: `CrosslayerPortal/contracts/functionCalls/*`	

Description
Cross-chain message calls that are requested on a low-fee, low-latency network could facilitate a DoS, preventing other users from interacting with the system.

If a user, through the `MsgSender` contract, sent numerous cross-chain message call requests, the relayer would have to act upon the emitted events regardless of whether they were legitimate or part of a DoS attack.

Exploit Scenario
Eve creates a *theoretically* infinite series of transactions on Arbitrum, a low-fee, low-latency network. The internal queue of the relayer is then filled with numerous malicious transactions. Alice requests a cross-chain message call; however, because the relayer must handle many of Eve's transactions first, Alice has to wait an undefined amount of time for her transaction to be executed.

Recommendations
Short term, create multiple queues that work across the various chains to mitigate this DoS risk.

Long term, analyze the implications of the ability to create numerous message calls on low-fee networks and its impact on relayer performance.

11. Missing validation in takeFees function

Difficulty: **Medium**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-CMP-11
Target: `CrossLayerPortal/contracts/core/MosaicVault.sol`	

Description

The `takeFees` function takes `calldata` from relayers and contract owners but does not validate the `baseFee` value passed to it. This lack of validation allows relayers and owners to set arbitrarily large `baseFee` values.

To withdraw funds from the `MosaicVault` contract, a relayer or owner invokes the `withdrawTo` function on the contract, passing in `_withdrawData` `calldata`.

```
/// @notice method called by the relayer to release transfer funds
/// @param _accountTo eth address to send the withdrawal tokens
/// @param _amount amount of token in
/// @param _tokenIn address of the token in
/// @param _tokenOut address of the token out
/// @param _amountOutMin minimum amount out user wants
/// @param _withdrawData set of data for withdraw
/// @param _data additional data required for each AMM implementation
function withdrawTo(
    address _accountTo,
    uint256 _amount,
    address _tokenIn,
    address _tokenOut,
    uint256 _amountOutMin,
    WithdrawData calldata _withdrawData,
    bytes calldata _data
) external override {
    uint256 withdrawAmount = _withdraw(
        _accountTo,
        _amount,
        _tokenIn,
        _tokenOut,
        _amountOutMin,
        _withdrawData,
        _data
    );

    emit WithdrawalCompleted(
        _accountTo,
        _amount,
        withdrawAmount,
        _tokenOut,
        _withdrawData.id,
        _withdrawData.amountToSwapToNative > 0
    );
}
```

Figure 11.1: The `withdrawTo` function in `MosaicVault:359-394`

The internal `_withdraw` function verifies that the caller is either a relayer or owner and invokes the `takeFees` function.

```
/// @dev internal function called by `withdrawLiquidity` and `withdrawTo`
/// @param _accountTo eth address to send the withdrawal tokens
/// @param _amount amount of token in
/// @param _tokenIn address of the token in
/// @param _tokenOut address of the token out
/// @param _amountOutMin minimum amount out user wants
/// @param _withdrawData set of data for withdraw
/// @param _data additional data required for each AMM implementation
function _withdraw(
    address _accountTo,
    uint256 _amount,
    address _tokenIn,
    address _tokenOut,
    uint256 _amountOutMin,
    WithdrawData calldata _withdrawData,
    bytes calldata _data
)
    internal
    onlyWhitelistedToken(_tokenIn)
    validAddress(_tokenOut)
    nonReentrant
    onlyOwnerOrRelayer
    whenNotPaused
    returns (uint256 withdrawAmount)
{
    IMosaicHolding mosaicHolding = IMosaicHolding(vaultConfig.getMosaicHolding());
    require(hasBeenWithdrawn[_withdrawData.id] == false, "ERR: ALREADY WITHDRAWN");
    if (_tokenOut == _tokenIn) {
        require(
            mosaicHolding.getTokenLiquidity(_tokenIn, _withdrawData.investmentStrategies) >=
                _amount,
            "ERR: VAULT BAL"
        );
    }
    uint256 amountToNative = _withdrawData.amountToSwapToNative;
    if (amountToNative > 0) {
        require(vaultConfig.ableToPerformSmallBalanceSwap(), "ERR: UNABLE TO SWAP TO
NATIVE");
    }
    require(amountToNative <= _amount, "ERR: AMOUNT TO NATIVE TOO HIGH");
    hasBeenWithdrawn[_withdrawData.id] = true;
    lastWithdrawID = _withdrawData.id;

    withdrawAmount = _takeFees(
        _tokenIn,
        _amount,
        _accountTo,
        _withdrawData.id,
        _withdrawData.baseFee,
        _withdrawData.feePercentage
    );
```

Figure 11.2: The `_withdraw` function in `MosaicVault:396-445`

The contract checks that the `feePercentage` value is between the `MosaicVault`-configured minimum and maximum. However, there is no validation of the `baseFee` value. This lack of validation allows the caller to specify an arbitrarily large `baseFee` amount.

```
/// @dev internal function called to calculate the on-chain fees
/// @param _token address of the token in
/// @param _amount amount of token in
/// @param _accountTo address who will receive the withdrawn liquidity
/// @param _withdrawRequestId id of the withdraw request
/// @param _baseFee base fee of the withdraw
/// @param _feePercentage fee percentage of the withdraw
function _takeFees(
    address _token,
    uint256 _amount,
    address _accountTo,
    bytes32 _withdrawRequestId,
    uint256 _baseFee,
    uint256 _feePercentage
) private returns (uint256) {
    if (_baseFee > 0) {
        IMosaicHolding(vaultConfig.getMosaicHolding()).transfer(_token, relayer, _baseFee);
    }
    uint256 fee = 0;
    if (_feePercentage > 0) {
        require(
            _feePercentage >= vaultConfig.minFee() && _feePercentage <=
vaultConfig.maxFee(),
            "ERR: FEE PERCENTAGE OUT OF RANGE"
        );

        fee = FeeOperations.getFeeAbsolute(_amount, _feePercentage);

        IMosaicHolding(vaultConfig.getMosaicHolding()).transfer(
            _token,
            vaultConfig.getMosaicHolding(),
            fee
        );
    }

    uint256 totalFee = _baseFee + fee;
    require(totalFee < _amount, "ERR: FEE EXCEEDS AMOUNT");
    if (totalFee > 0) {
        emit FeeTaken(
            msg.sender,
            _accountTo,
            _token,
            _amount,
            fee,
            _baseFee,
            fee + _baseFee,
            _withdrawRequestId
        );
    }
    return _amount - totalFee;
}
```

Figure 11.3: The `takeFees` function in `MosaicVault:548-597`

Exploit Scenario

Alice, a user, submits a withdrawal request, assuming a `baseFee` of 5 DAI. When executing the withdrawal, the relayer sets the `baseFee` to 100 DAI, significantly decreasing the amount of funds Alice receives through her withdrawal.

Recommendations

Short term, add sufficient data validation of the `baseFee` value to ensure that relayers cannot set it to arbitrarily large values.

Long term, use Echidna to test the system invariants and ensure that they hold.

12. Unimplemented getAmountsOut function in Balancer V2

Severity: **Medium**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-CMP-12
Target: `CrosslayerPortal/contracts/protocols/balancer/BalancerVaultV2Wrapper.sol#L43-L50`	

Description

The `getAmountsOut` function in the `BalancerV2Wrapper` contract is unimplemented.

The purpose of the `getAmountsOut()` function, shown in figure 12.1, is to allow users to know the amount of funds they will receive when executing a swap. Because the function does not invoke any functions on the Balancer Vault, a user must actually perform a swap to determine the amount of funds he or she will receive:

```
function getAmountsOut(
    address,
    address,
    uint256,
    bytes calldata
) external pure override returns (uint256) {
    return 0;
}
```

Figure 12.1: The `getAmountsOut` function in `BalancerVaultV2Wrapper:43-50`

Exploit Scenario

Alice, a user of the Composable Finance vaults, wants to swap 100 USDC for DAI on Balancer. Because the `getAmountsOut` function is not implemented, she is unable to determine how much DAI she will receive before executing the swap.

Recommendations

Short term, implement the `getAmountsOut` function and have it call the `queryBatchSwap` function on the Balancer Vault.

Long term, add unit tests for all functions to test all flows. Unit tests will detect incorrect function behavior.

28. Lack of events for critical operations

Severity: **Medium**	Difficulty: **Low**
Type: Auditing and Logging	Finding ID: TOB-CMP-28

Target: `CrosslayerPortal/contracts/core/MosaicVault.sol,CrosslayerPortal/contracts/core/BridgeAggregator.sol,CrosslayerPortal/contracts/nfts/MosaicNFT.sol`

Description

Several critical operations do not trigger events. As a result, it will be difficult to review the correct behavior of the contracts once they have been deployed.

For example, the `setRelayer` function, which is called in the `MosaicVault` contract to set the relayer address, does not emit an event providing confirmation of that operation to the contract's caller (figure 28.1).

```
function setRelayer(address _relayer) external override onlyOwner {
    relayer = _relayer;
}
```

Figure 28.1: The `setRelayer()` function in `MosaicVault:80-82`

Without events, users and blockchain-monitoring systems cannot easily detect suspicious behavior.

Exploit Scenario

Eve, an attacker, is able to take ownership of the `MosaicVault` contract. She then sets a new relayer address. Alice, a Composable Finance team member, is unaware of the change and does not raise a security incident.

Recommendations

Short term, add events for all critical operations that result in state changes. Events aid in contract monitoring and the detection of suspicious behavior.

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

30. Insufficient protection of sensitive information

Severity: **Medium**	Difficulty: **High**
Type: Configuration	Finding ID: TOB-CMP-30
Target: `CrosslayerPortal/env`, `bribe-protocol/hardhat.config.ts`	

Description

Sensitive information such as a mnemonic seed phrase and API keys is stored in process environments and environment files. This method of storage could make it easier for an attacker to compromise the information; compromise of the seed phrase, for example, could enable an attacker to gain owner privileges and steal funds from the protocol.

The following portion of the `bribe-protocol/hardhat.config.ts` file uses a mnemonic seed phrase from the process environment:

```
  mainnet: {

     accounts: {

       mnemonic: process.env.MNEMONIC || "",

     },

     url: "https://eth-mainnet.alchemyapi.io/v2/[redacted]",

   },

}
```

Figure 30.1: Use of a mnemonic seed phrase from the process environment in the `bribe-protocol/hardhat.config.ts` file

The following portion of the `CrosslayerPortal/env/mainnet.env` file exposes Etherscan and Alchemy API keys:

```
NETWORK_NAME=mainnet
```

```
CHAIN_ID = 1



## API

BLOCK_EXPLORER_API_KEY = [redacted]

ALCHEMY_API=https://eth-mainnet.alchemyapi.io/v2/[redacted]
```

Figure 30.2: Exposed API keys in the `CrosslayerPortal/env/mainnet.env` *file*

If this repository is ever made public, the Git commit history will retain these API keys in plaintext. Access to the Alchemy API key, for example, would allow an attacker to launch a DoS attack on the application's front end.

Exploit Scenario

Alice has the owner's mnemonic seed phrase stored in her process environment. Eve, an attacker, gains access to Alice's device and extracts the seed phrase from it. Eve uses the owner key to steal all funds.

Recommendations

Short term, use a hardware security module to ensure that none of the keys can be extracted. Additionally, avoid using hard-coded secrets and committing environment files to version control systems.

Long term, take the following steps:

- Move key material from environment variables to a dedicated secret management system with trusted computing abilities. The best options for such a system are the Google Cloud Key Management System and Hashicorp Vault (with hardware security module backing).

- Determine the business risk that would result from a lost or stolen key and develop disaster recovery and business continuity policies to be implemented in such a case.

5. Accrued interest is not attributable to the underlying investor on-chain

Severity: **Low**	Difficulty: **Undetermined**
Type: Data Validation	Finding ID: TOB-CMP-5
Target: `CrosslayerPortal/contracts/core/MosaicHolding.sol`	

Description

When an investor earns interest-bearing tokens by lending funds through Mosaic's investment strategies, the tokens are not directly attributed to the investor by on-chain data.

The `claim()` function, which can be called only by the owner of the `MosaicHolding` contract, is defined in the contract and used to redeem interest-bearing tokens from protocols such as Aave and Compound (figure 5.1). The underlying tokens of these protocols' lending pools are provided by users who are interacting with the Mosaic system and wish to earn rewards on their idle funds.

```
function claim(address _investmentStrategy, bytes calldata _data)
    external
    override
    onlyAdmin
    validAddress(_investmentStrategy)
{
    require(investmentStrategies[_investmentStrategy], "ERR: STRATEGY NOT SET");
    address rewardTokenAddress =
IInvestmentStrategy(_investmentStrategy).claimTokens(_data);
    emit TokenClaimed(_investmentStrategy, rewardTokenAddress);
}
```

Figure 5.1: The `claim` function in `MosaicHolding:270-279`

During the execution of `claim()`, the internal `claimTokens()` function calls into the `AaveInvestmentStrategy`, `CompoundInvestmentStrategy`, or `SushiswapLiquidityProvider` contract, which effectively transfers its balance of the interest-bearing token directly to the `MosaicHolding` contract. Figure 5.2 shows the `claimTokens()` function call in `AaveInvestmentStrategy`.

```
 function claimTokens(bytes calldata data) external override onlyInvestor returns (address)
{
    address token = abi.decode(data, (address));
    ILendingPool lendingPool = ILendingPool(lendingPoolAddressesProvider.getLendingPool());
    DataTypes.ReserveData memory reserve = lendingPool.getReserveData(token);

    IERC20Upgradeable(reserve.aTokenAddress).safeTransfer(
        mosaicHolding,
        IERC20Upgradeable(reserve.aTokenAddress).balanceOf(address(this))
    );
    return reserve.aTokenAddress;
}
```

Figure 5.2: The `claimTokens` function in `AaveInvestmentStrategy:58-68`

However, there is no identifiable mapping or data structure attributing a percentage of those rewards to a given user. The off-chain relayer service is responsible for holding such mappings and rewarding users with the interest they have accrued upon withdrawal (see the relayer bot assumptions in the Project Coverage section).

Exploit Scenario

Investors Alice and Bob, who wish to earn interest on their idle USDC, decide to use the Mosaic system to provide loans. Mosaic invests their money in Aave's lending pool for USDC. However, there is no way for the parties to discern their ownership stakes in the lending pool through the smart contract logic. The owner of the contract decides to call the `claim()` function and redeem all aUSDC associated with Alice's and Bob's positions. When Bob goes to withdraw his funds, he has to *trust* that the relayer will send him his claim on the aUSDC without any on-chain verification.

Recommendations

Short term, consider implementing a way to identify the amount of each investor's stake in a given investment strategy. Currently, the relayer is responsible for tracking all rewards.

Long term, review the privileges and responsibilities of the relayer and architect a more robust solution for managing investments.

6. User funds can become trapped in nonstandard token contracts

Severity: **Low**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-CMP-6
Target: `CrosslayerPortal/contracts/functionCalls/MsgReceiver.sol`	

Description

If a user's funds are transferred to a token contract that violates the ERC20 standard, the funds may become permanently trapped in that token contract.

In the `MsgReceiver` contract, there are six calls to the `transfer()` function. See figure 6.1 for an example.

```
function approveERC20TokenAndForwardCall(
    uint256 _feeAmount,
    address _feeToken,
    address _feeReceiver,
    address _token,
    uint256 _amount,
    bytes32 _id,
    address _contract,
    bytes calldata _data
) external payable onlyOwnerOrRelayer returns (bool success, bytes memory returnData) {
    require(
        IMsgReceiverFactory(msgReceiverFactory).whitelistedFeeTokens(_feeToken),
        "Fee token is not whitelisted"
    );
    require(!forwarded[_id], "call already forwarded");
    //approve tokens to _contract
    IERC20(_token).safeIncreaseAllowance(_contract, _amount);
    // solhint-disable-next-line avoid-low-level-calls
    (success, returnData) = _contract.call{value: msg.value}(_data);
    require(success, "Failed to forward function call");

    uint256 balance = IERC20(_feeToken).balanceOf(address(this));
    require(balance >= _feeAmount, "Not enough tokens for the fee");
    forwarded[_id] = true;
    IERC20(_feeToken).transfer(_feeReceiver, _feeAmount);
}
```

Figure 6.1: The `approveERC20TokenAndForwardCall` function in `MsgReceiver:98-123`

When implemented in accordance with the ERC20 standard, the `transfer()` function returns a boolean indicating whether a transfer operation was successful. However, tokens that implement the ERC20 interface incorrectly may not return `true` upon a successful transfer, in which case the transaction will revert and the user's funds will be locked in the token contract.

Exploit Scenario
Alice, the owner of the `MsgReceiverFactory` contract, adds a fee token that is controlled by Eve. Eve's token contract incorrectly implements the ERC20 interface. Bob interacts with `MsgReceiver` and calls a function that executes a transfer to `_feeReceiver`, which is controlled by Eve. Because Eve's fee token contract does not provide a return value, Bob's transfer reverts.

Recommendations
Short term, use `safeTransfer()` for token transfers and use the `SafeERC20` library for interactions with ERC20 token contracts.

Long term, develop a process for onboarding new fee tokens. Review our Token Integration Checklist for guidance on the onboarding process.

References

- Missing Return Value Bug

13. Use of MsgReceiver to check _feeToken status leads to unnecessary gas consumption

Severity: **Low**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-CMP-13
Target: `CrosslayerPortal/contracts/functionCalls/MsgReceiver.sol`	

Description

Checking the whitelist status of a token only on the receiving end of a message call can lead to excessive gas consumption.

As part of a cross-chain message call, all functions in the `MsgReceiver` contract check whether the token used for the payment to `_feeReceiver` (`_feeToken`) is a whitelisted token (figure 13.1). Tokens are whitelisted by the owner of the `MsgReceiverFactory` contract.

```
function approveERC20TokenAndForwardCall(
    uint256 _feeAmount,
    address _feeToken,
    address _feeReceiver,
    address _token,
    uint256 _amount,
    bytes32 _id,
    address _contract,
    bytes calldata _data
) external payable onlyOwnerOrRelayer returns (bool success, bytes memory
returnData) {
    require(
        IMsgReceiverFactory(msgReceiverFactory).whitelistedFeeTokens(_feeToken),
        "Fee token is not whitelisted"
    );
    require(!forwarded[_id], "call already forwarded");
    //approve tokens to _contract
    IERC20(_token).safeIncreaseAllowance(_contract, _amount);
    // solhint-disable-next-line avoid-low-level-calls
    (success, returnData) = _contract.call{value: msg.value}(_data);
    require(success, "Failed to forward function call");
    uint256 balance = IERC20(_feeToken).balanceOf(address(this));
    require(balance >= _feeAmount, "Not enough tokens for the fee");
    forwarded[_id] = true;
    IERC20(_feeToken).transfer(_feeReceiver, _feeAmount);
}
```

Figure 13.1: The `approveERC20TokenAndForwardCall` function in `MsgReceiver:98-123`

This validation should be performed before the `MsgSender` contract emits the related event (figure 13.2). This is because the relayer will act upon the emitted event on the receiving chain regardless of whether `_feeToken` is set to a whitelisted token.

```
function registerCrossFunctionCallWithTokenApproval(
    uint256 _chainId,
    address _destinationContract,
    address _feeToken,
    address _token,
    uint256 _amount,
    bytes calldata _methodData
)
    external
    override
    nonReentrant
    onlyWhitelistedNetworks(_chainId)
    onlyUnpausedNetworks(_chainId)
    whenNotPaused
{
    bytes32 id = _generateId();

    //shouldn't happen
    require(hasBeenForwarded[id] == false, "Call already forwarded");
    require(lastForwardedCall != id, "Forwarded last time");

    lastForwardedCall = id;
    hasBeenForwarded[id] = true;

    emit ForwardCallWithTokenApproval(
        msg.sender,
        id,
        _chainId,
        _destinationContract,
        _feeToken,
        _token,
        _amount,
        _methodData
    );
}
```

Figure 13.2: The `registerCrossFunctionCallWithTokenApproval` function in `MsgSender:169-203`

Exploit Scenario

On Arbitrum, a low-fee network, Eve creates a *theoretically* infinite series of transactions to be sent to `MsgSender`, with `_feeToken` set to a token that she knows is not whitelisted. The relayer relays the series of message calls to a `MsgReceiver` contract on Ethereum, a high-fee network, and all of the transactions revert. However, the relayer has to pay the intrinsic gas cost for each transaction, with no repayment, while allowing its internal queue to be filled up with malicious transactions.

Recommendations

Short term, move the logic for token whitelisting and validation to the `MsgSender` contract.

Long term, analyze the implications of the ability to create numerous message calls on low-fee networks and its impact on relayer performance.

14. Active liquidity providers can set arbitrary _tokenOut values when withdrawing liquidity	
Severity: **Low**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-CMP-14
Target: `CrosslayerPortal/contracts/core/MosaicVault.sol`	

Description

An active liquidity provider (LP) can move his or her liquidity into any token, even one that the LP controls.

When a relayer acts upon a `WithdrawRequest` event triggered by an *active* LP, the `MosaicVault` contract checks only that the address of `_tokenOut` (the token being requested) is not the zero address (figure 14.1). Outside of that constraint, `_tokenOut` can effectively be set to any token, even one that might have vulnerabilities.

```
function _withdraw(
    address _accountTo,
    uint256 _amount,
    address _tokenIn,
    address _tokenOut,
    uint256 _amountOutMin,
    WithdrawData calldata _withdrawData,
    bytes calldata _data
)
    internal
    onlyWhitelistedToken(_tokenIn)
    validAddress(_tokenOut)
    nonReentrant
    onlyOwnerOrRelayer
    whenNotPaused
    returns (uint256 withdrawAmount)
```

Figure 14.1: The signature of the `_withdraw` function in `MosaicVault:404-419`

This places the burden of ensuring the swap's success on the decentralized exchange, and, as the application grows, can lead to unintended code behavior.

Exploit Scenario
Eve, a malicious active LP, is able to trigger undefined behavior in the system by setting `_tokenOut` to a token that is vulnerable to exploitation.

Recommendations
Short term, analyze the implications of allowing `_tokenOut` to be set to an arbitrary token.

Long term, validate the assumptions surrounding the lack of limits on `_tokenOut` as the codebase grows, and review our Token Integration Checklist to identify any related pitfalls.

References
- imBTC Uniswap Hack

15. Withdrawal assumptions may lead to transfers of an incorrect token

Severity: **Low**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-CMP-15
Target: `apyhunter-tricrypto/contracts/tricrypto/CurveTricryptoStrategy.sol`	

Description

The `CurveTricryptoStrategy` contract manages liquidity in Curve pools and facilitates transfers of tokens between chains. While it is designed to work with one curve vault, the vault can be set to an arbitrary pool. Thus, the contract should not make assumptions regarding the pool without validation.

Each pool contains an array of tokens specifying the tokens to withdraw from that pool. However, when the vault address is set in the constructor of `CurveTricryptoConfig`, the pool's address is not checked against the TriCrypto pool's address. The token at index 2 in the `coins` array is assumed to be wrapped ether (WETH), as indicated by the code comment shown in figure 15.1. If the configuration is incorrect, a different token may be unintentionally transferred.

```
if (unwrap) {
    //unwrap LP into weth
    transferredToken = tricryptoConfig.tricryptoLPVault().coins(2);
[...]
```

Figure 15.1: Part of the `transferLPs` function in `CurveTricryptoStrategy.sol:377-379`

Exploit Scenario

The Curve pool array, `coins`, stores an address other than that of WETH in index 2. As a result, a user mistakenly sends the wrong token in a transfer.

Recommendations

Short term, have the constructor of `CurveTricryptoConfig` or the `transferLPs` function validate that the address of `transferredToken` is equal to the address of WETH.

Long term, validate data from external contracts, especially data involved in the transfer of funds.

16. Improper validation of Chainlink data	
Severity: **Low**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-CMP-16
Target: `v2-sdk-contracts/contracts/libraries/oracles/ChainlinkLib.sol`	

Description

The `latestRoundData` function returns a signed integer that is coerced to an unsigned integer without checking that the value is positive. An overflow (e.g., `uint(-1)`) would result in drastic misrepresentation of the price and unexpected behavior.

In addition, `ChainlinkLib` does not ensure the completeness or recency of round data, so pricing data may not reflect recent changes. It is best practice to define a window in which data is considered sufficiently recent (e.g., within a minute of the last update) by comparing the block timestamp to `updatedAt`.

```
(, int256 price, , , ) = _aggregator.latestRoundData();
return uint256(price);
```

Figure 16.1: Part of the `getCurrentTokenPrice` function in `ChainlinkLib.sol:113-114`

Recommendations

Short term, have `latestRoundData` and similar functions verify that values are non-negative before converting them to unsigned integers, and add an invariant—`require(updatedAt != 0 && answeredInRound == roundID)`—to ensure that the round has finished and that the pricing data is from the current round.

Long term, define a minimum update threshold and add the following check: `require((block.timestamp - updatedAt <= minThreshold) && (answeredInRound == roundID))`.

25. Incorrect safeIncreaseAllowance() amount can cause invest() calls to revert	
Severity: **Low**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-CMP-25
Target: `CrosslayerPortal/contracts/protocols/sushiswap/SushiswapLiquidityProvider.sol`	

Description

Calls to make investments through Sushiswap can revert because the `sushiSwapRouter` may not have the token allowances needed to fulfill the requests.

The owner of the `MosaicHolding` contract is responsible for investing user-deposited funds in investment strategies. The contract owner does this by calling the contract's `invest()`function, which then calls `makeInvestment()` on the investment strategy meant to receive the funds (figure 25.1).

```
function invest(
    IInvestmentStrategy.Investment[] calldata _investments,
    address _investmentStrategy,
    bytes calldata _data
) external override onlyAdmin validAddress(_investmentStrategy) {
    require(investmentStrategies[_investmentStrategy], "ERR: STRATEGY NOT SET");
    uint256 investmentsLength = _investments.length;
    address contractAddress = address(this);
    for (uint256 i; i < investmentsLength; i++) {
        IInvestmentStrategy.Investment memory investment = _investments[i];
        require(investment.amount != 0 && investment.token != address(0), "ERR:
TOKEN AMOUNT");
        IERC20Upgradeable token = IERC20Upgradeable(investment.token);
        require(token.balanceOf(contractAddress) >= investment.amount, "ERR:
BALANCE");
        token.safeApprove(_investmentStrategy, investment.amount);
    }

    uint256 mintedTokens = IInvestmentStrategy(_investmentStrategy).makeInvestment(
        _investments,
        _data
    );
    emit FoundsInvested(_investmentStrategy, msg.sender, mintedTokens);
}
```

Figure 25.1: The `invest` function in `MosaicHolding:190-211`

To deposit funds into the `SushiswapLiquidityProvider` investment strategy, the contract must increase the `sushiSwapRouter`'s approval limits to account for the `tokenA` and `tokenB` amounts to be transferred.

However, `tokenB`'s approval limit is increased only to the amount of the `tokenA` investment (figure 25.2).

```
function makeInvestment(Investment[] calldata _investments, bytes calldata _data)
    external
    override
    onlyInvestor
    nonReentrant
    returns (uint256)
{
    Investment memory investmentA = _investments[0];
    Investment memory investmentB = _investments[1];
    IERC20Upgradeable tokenA = IERC20Upgradeable(investmentA.token);
    IERC20Upgradeable tokenB = IERC20Upgradeable(investmentB.token);

    tokenA.safeTransferFrom(msg.sender, address(this), investmentA.amount);
    tokenB.safeTransferFrom(msg.sender, address(this), investmentB.amount);

    tokenA.safeIncreaseAllowance(address(sushiSwapRouter), investmentA.amount);
    tokenB.safeIncreaseAllowance(address(sushiSwapRouter), investmentA.amount);

    (uint256 deadline, uint256 minA, uint256 minB) = abi.decode(
        _data,
        (uint256, uint256, uint256)
    );
    (, , uint256 liquidity) = sushiSwapRouter.addLiquidity(
        investmentA.token,
        investmentB.token,
        investmentA.amount,
        investmentB.amount,
        minA,
        minB,
        address(this),
        deadline
    );
    return liquidity;
}
```

Figure 25.2: The `makeInvestment` function in `SushiswapLiquidityProvider:52-85`

If the amount of `tokenB` to be deposited is greater than that of `tokenA`, `sushiSwapRouter` will fail to transfer the tokens, and the transaction will revert.

Exploit Scenario

Alice, the owner of the `MosaicHolding` contract, wishes to invest liquidity in a Sushiswap liquidity pool. The amount of the `tokenB` investment is greater than that of `tokenA`. The `sushiSwapRouter` does not have the right token allowances for the transaction, and the investment request fails.

Recommendations

Short term, change the amount value used in the `safeIncreaseAllowance()` call from `investmentA.amount` to `investmentB.amount`.

Long term, review the codebase to identify similar issues. Additionally, create a more extensive test suite capable of testing edge cases that may invalidate system assumptions.

17. Incorrect check of token status in the providePassiveLiquidity function

Severity: **Informational**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-CMP-17
Target: `CrosslayerPortal/contracts/core/MosaicVault.sol`	

Description

A passive LP can provide liquidity in the form of a token that is not whitelisted.

The `providePassiveLiquidity()` function in `MosaicVault` is called by users who wish to participate in the Mosaic system as passive LPs. As part of the function's execution, it checks whether there is a `ReceiptToken` associated with the `_tokenAddress` input parameter (figure 17.1). This is equivalent to checking whether the token is whitelisted by the system.

```
function providePassiveLiquidity(uint256 _amount, address _tokenAddress)
    external
    payable
    override
    nonReentrant
    whenNotPaused
{
    require(_amount > 0 || msg.value > 0, "ERR: AMOUNT");
    if (msg.value > 0) {
        require(
            vaultConfig.getUnderlyingIOUAddress(vaultConfig.wethAddress()) !=
address(0),
            "ERR: WETH NOT WHITELISTED"
        );
        _provideLiquidity(msg.value, vaultConfig.wethAddress(), 0);
    } else {
        require(_tokenAddress != address(0), "ERR: INVALID TOKEN");
        require(
            vaultConfig.getUnderlyingIOUAddress(_tokenAddress) != address(0),
            "ERR: TOKEN NOT WHITELISTED"
        );
        _provideLiquidity(_amount, _tokenAddress, 0);
    }
}
```

Figure 17.1: The `providePassiveLiquidity` function in `MosaicVault:127-149`

However, `providePassiveLiquidity()` uses an incorrect function call to check the whitelist status. Instead of calling `getUnderlyingReceiptAddress()`, it calls `getUnderlyingIOUAddress()`. The same issue occurs in checks of WETH deposits.

Exploit Scenario

Eve decides to deposit liquidity in the form of a token that is whitelisted only for active LPs. The token provides a higher yield than the tokens whitelisted for passive LPs. This may enable Eve to receive a higher annual percentage yield on her deposit than other passive LPs in the system receive on theirs.

Recommendations

Short term, change the function called to validate `tokenAddress` and `wethAddress` from `getUnderlyingIOUAddress()` to `getUnderlyingReceiptAddress()`.

Long term, take the following steps:

- Review the codebase to identify similar errors.
- Consider whether the assumption that the same tokens will be whitelisted for both passive and active LPs will hold in the future.
- Create a more extensive test suite capable of testing edge cases that may invalidate system assumptions.

18. Solidity compiler optimizations can be problematic	
Severity: **Informational**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-CMP-18
Target: `CrosslayerPortal`, `apy-hunter`	

Description

The Composable Finance 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 Composable Finance contracts.

Recommendations

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

Long term, monitor the development and adoption of Solidity compiler optimizations to assess their maturity.

19. Lack of contract documentation	
Severity: **Informational**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-CMP-19
Target: Throughout the code	

Description

The codebases lack code documentation, high-level descriptions, and examples, making the contracts difficult to review and increasing the likelihood of user mistakes.

The `CrosslayerPortal` codebase would benefit from additional documentation, including on the following:

- The logic responsible for setting the roles in the core and the reason for the manipulation of indexes
- The incoming function arguments and the values used on source chains and destination chains
- The arithmetic involved in reward calculations and the relayer's distribution of tokens
- The checks performed by the off-chain components, such as the relayer and the rebalancing bot
- The third-party integrations
- The rebalancing arithmetic and calculations

There should also be clear NatSpec documentation on every function, identifying the unit of each variable, the function's intended use, and the function's safe values.

The documentation should include all expected properties and assumptions relevant to the aforementioned aspects of the codebase.

Recommendations

Short term, review and properly document the aforementioned aspects of the codebase.

Long term, consider writing a formal specification of the protocol.

21. Unnecessary complexity due to interactions with native and smart contract tokens

Severity: **Informational**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-CMP-21
Target: `apyhunter-tricrypto/contracts/sushiswap/SushiSlpStrategy.sol`	

Description

The Composable Finance code is needlessly complex and has excessive branching. Its complexity largely results from the integration of both ERC20s and native tokens (i.e., ether). Creating separate functions that convert native tokens to ERC20s and then interact with functions that must receive ERC20 tokens (i.e., implementing separation of concerns) would drastically simplify and optimize the code.

This complexity is the source of many bugs and increases the gas costs for all users whether or not they need to distinguish between ERC20s and ether. It is best practice to make components as small as possible and to separate helpful but noncritical components into periphery contracts. This reduces the attack surface and improves readability.

Figure 21.1 shows an example of complex code.

```
if (tempData.isSlp) {
    IERC20(sushiConfig.slpToken()).safeTransfer(
        msg.sender,
        tempData.slpAmount
    );
    [...]
} else {
    //unwrap and send the right asset
    [...]

    if (tempData.isEth) {
        [...]
    } else {
        IERC20(sushiConfig.wethToken()).safeTransfer(
```

Figure 21.1: Part of the `withdraw` function in `SushiSlpStrategy.sol:L180-L216`

Recommendations

Short term, remove the native ether interactions and use WETH instead.

Long term, minimize the function complexity by breaking functions into smaller units. Additionally, refactor the code with minimalism in mind and extend the core functionality into periphery contracts.

29. Use of legacy openssl version in CrosslayerPortal tests

Severity: **Informational**	Difficulty: **N/A**
Type: Patching	Finding ID: TOB-CMP-29
Target: `CrosslayerPortal`	

Description

The `CrosslayerPortal` project uses a legacy version of `openssl` to run tests. While this version is not exposed in production, the use of outdated security protocols may be risky (figure 29.1).

```
An unexpected error occurred:

Error: error:0308010C:digital envelope routines::unsupported
    at new Hash (node:internal/crypto/hash:67:19)
    at Object.createHash (node:crypto:130:10)
    at hash160
(~/CrosslayerPortal/node_modules/ethereum-cryptography/vendor/hdkey-without-crypto.js:249:21
)
    at HDKey.set
(~/CrosslayerPortal/node_modules/ethereum-cryptography/vendor/hdkey-without-crypto.js:50:24)
    at Function.HDKey.fromMasterSeed
(~/CrosslayerPortal/node_modules/ethereum-cryptography/vendor/hdkey-without-crypto.js:194:20
)
    at deriveKeyFromMnemonicAndPath
(~/CrosslayerPortal/node_modules/hardhat/src/internal/util/keys-derivation.ts:21:27)
    at derivePrivateKeys
(~/CrosslayerPortal/node_modules/hardhat/src/internal/core/providers/util.ts:29:52)
    at normalizeHardhatNetworkAccountsConfig
(~/CrosslayerPortal/node_modules/hardhat/src/internal/core/providers/util.ts:56:10)
    at createProvider
(~/CrosslayerPortal/node_modules/hardhat/src/internal/core/providers/construction.ts:78:59)
    at
~/CrosslayerPortal/node_modules/hardhat/src/internal/core/runtime-environment.ts:80:28 {
  opensslErrorStack: [ 'error:03000086:digital envelope routines::initialization error' ],
  library: 'digital envelope routines',
  reason: 'unsupported',
  code: 'ERR_OSSL_EVP_UNSUPPORTED'
}
```

Figure 29.1: Errors flagged in `npx hardhat` testing

Recommendations

Short term, refactor the code to use a new version of `openssl` to prevent the exploitation of `openssl` vulnerabilities.

Long term, avoid using outdated or legacy versions of dependencies.

22. Commented-out and unimplemented conditional statements	
Severity: **Undetermined**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-CMP-22
Target: `apyhunter-tricrypto/contracts/sushiswap/SushiSlpStrategy.sol`	

Description

The Composable Finance system does not define or document the expected behavior of certain conditional statements. It is difficult to assess the correctness or security of the code without an understanding of its expected behavior.

For example, it is unclear why one `if` statement is commented out.

```
// if (!_isSenderKeeperOrOwner(msg.sender)) {
    totalAmountOfLPs = totalAmountOfLPs - tempData.slpAmount;
// }
```

Figure 22.1: Part of the `withdraw` function in `SushiSlpStrategy.sol:L404`

Another `if` statement does not define what should happen when the condition evaluates to `true`.

```
if (_isSenderKeeperOrOwner(msg.sender) && _isReceiveBackLP) {}
```

Figure 22.2: Part of the `_deposit` function in `SushiSlpStrategy.sol:L177-L179`

This pattern is error-prone and bad practice.

Recommendations

Short term, fully implement the necessary logic and remove commented-out code.

Long term, remove the ambiguity surrounding the code and conditional branches by clearly documenting and testing the system’s functions.

23. Error-prone NFT management in the Summoner contract

Severity: **Undetermined**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-CMP-23
Target: `CrosslayerPortal`	

Description

The `Summoner` contract's ability to hold NFTs in a number of states may create confusion regarding the contracts' states and the differences between the contracts.

For instance, the `Summoner` contract can hold the following kinds of NFTs:

- NFTs that have been pre-minted by Composable Finance and do not have metadata attached to them
- Original NFTs that have been locked by the `Summoner` for minting on the destination chain
- `MosaicNFT` wrapper tokens, which are copies of NFTs that have been locked and are intended to be minted on the destination chain

As the system is scaled, the number of NFTs held by the `Summoner`, especially the number of pre-minted NFTs, will increase significantly.

Recommendations

Simplify the NFT architecture; see the related recommendations in Appendix E.

Summary of Recommendations

The Composable Finance Mosaic infrastructure is a work in progress with multiple planned iterations. Trail of Bits recommends that Composable Finance address the findings detailed in this report and take the following additional steps prior to deployment:

- Simplify the architecture and reduce the system's complexity by building a minimal version of the smart contracts. For additional guidance, see the recommendations in Appendix D and Appendix E.
- Integrate proper (and running) unit tests into the system and a continuous integration pipeline, and test all "happy paths" and expected revert flows in the contracts prior to considering deployment.
- Write clear and concise documentation on the expected behavior of all functions in the system.
- Identify and analyze all system properties that are expected to hold.
- Use Echidna to test system properties. Ensure that these properties will always hold.
- Do not deploy the `MosaicVault` or `MosaicHolding` contract until another in-depth security review has been completed.

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 Calls	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

General

- **Avoid including redundant checks in function calls.** Identify variables that are subject to change and add checks for only those variables.
- **Create a helper library that is reusable across all contracts.** This library can include functions such as `validAddress` and `validAmount`.
- **Avoid copying storage variables into memory when they are read only once.** For example, use `myArray.length` instead of creating a new memory variable called `myArrayLength`.
- **Address the many function parameters that are unused.** Rather than leaving parameters unused to comply with the interface specifications, consider breaking apart the current interfaces into more specific ones.
- **Use Slither's built-in `slither-check-upgradeability` tool to help verify the storage layout when developing upgradeable contracts.** It can help to catch common upgradeability issues.
- **Avoid performing equality comparisons on booleans.** The statement `require(isLocked)` is equivalent to `require(isLocked == true)`.
- **To reduce gas costs, make variables immutable wherever possible.**
- **Use the `SafeERC20` library wherever possible.** See TOB-CMP-6.
- **Avoid using a loop for each withdrawal, transfer, and investment operation.** Consider performing incremental bookkeeping during various process flows instead of using loops.

`MosaicVault`

- **Add `_withdrawData.baseFee` and `_withdrawData.feePercentage` to the `WithdrawalCompleted` event emitted at the end of the `withdrawTo()` function call.**

`MosaicHolding`

- **Address the typo in the `FoundsInvested` event by changing its name to `FundsInvested`.**

SDK

- **Avoid hard-coding function signatures in `OperationsLib.sol`; this practice is error-prone.** Use `abi.encodeWithSelector()` to encode function call selectors.

- **Ensure that every function that has `return` in its function signature provides a return value.**

- **Ensure that every function that provides a return value has `return` in its function signature.**

- **If a function's parameter is not modified during the function's execution, use `calldata` instead of `memory`.** This will result in gas savings.

NFTs

- **Avoid creating private data structures and subsequently writing getter functions for them.** Consider making data structures public. That way, the getter functionality will be provided automatically.

- **Use a more efficient naming convention for the types of NFT IDs that exist in the Mosaic ecosystem.**

`functionCalls`

- **Reuse functions that already implement required logic instead of rewriting it.** In the `MsgReceiver` contract, `approveERC20TokenAndForwardCall` can call `safeIncreaseAllowance` and then `forwardCall`. The logic of `forwardCall` does not need to be re-implemented.

D. Architecture Flaws and Recommendations

Trail of Bits reviewed the overall architecture of the `MosaicVault` and the associated contracts. This appendix provides recommendations on architecture improvements and code simplification.

Use Existing Trustless Infrastructure

The protocol proposes to use centralized bridges between systems with existing trustless bridges (e.g., Ethereum L1<>L2). Using a centralized solution creates additional risks and reduces the system's integrity, with unclear benefits.

Bridges like the Arbitrum bridge account for and mitigate many of the failure cases possible in the protocol, like those related to chain reorgs and finality. We recommend removing the relayer bot functionality, evaluating the existing solutions provided by other systems, and suggesting that users rely on trustless systems when they are available.

Improve Accounting Transparency

The system relies on the relayer to perform accurate off-chain accounting. This design makes it difficult for users to validate the payouts they receive—and that process will become more difficult once the vaults have been deployed across multiple blockchains. Implement the following recommendation to improve traceability:

- **Compute reward amounts on-chain instead of relying on off-chain computations.** The current practice of forwarding all interest-bearing tokens from a protocol (e.g., aTokens from Aave) to the `MosaicHolding` contract provides users no visibility on the amount of rewards they will receive upon withdrawal. The user must trust that the relayer bot will correctly calculate the reward amount between the time of the deposit and the time of the user’s exit.

Reduce Owner and Relayer Privileges

The protocol relies on the relayer to submit transactions on behalf of users. We recommend the following:

- **Allow users to specify a maximum gas cost for their transactions.** Because the relayer specifies the fee amount when withdrawing funds on users’ behalf, users do not actually know the gas costs they will incur until the relayer charges them.

- **Allow the relayer to execute a withdrawal only upon submission of a valid request.** The contracts currently allow the relayer to carry out a withdrawal at will, without checking whether a user has submitted a valid request.

Simplify the Code

The codebase contains many coding anti-patterns that make the code more difficult to read and review. The user approval flow, Composable Finance contracts, and external contract interactions seem over-engineered and could be made less gas intensive. Below we provide recommendations on ways to simplify the code.

- **Separate functions according to their purpose rather than including multiple branches in individual functions.** The current architecture commonly relies on a single public function as the entry point into the system and relies on branching conditions to decide subsequent logic. The code has high cyclomatic complexity as a consequence of this nested conditional logic.
    - Use Slither's `human-summary` printer to identify complex code.
- **Simplify the dependency structure and the logic shared by the repositories and npm packages.** This will minimize the inheritance tree and increase interoperability.
    - Use Slither's `inheritance-graph` printer to identify instances of complex inheritance.
- **Use `transfer` instead of `transferFrom` where appropriate.** When tokens are transferred from a contract itself, using `transfer` is more efficient and readable than using `transferFrom`.

Use Modifiers

The use of modifiers in Solidity can significantly improve code readability and aid in code analysis. We recommend the following:

- **Move all modifiers on functions that execute privileged actions into public / external functions (from internal functions).** This will make it easier to discern the modifiers protecting those functions.
- **Group code on the basis of user privilege or access levels.** This will improve the code's readability.

E. NFT Bridge Architecture Recommendations

Trail of Bits also reviewed the NFT bridge architecture. Given the bridge component's complexity, it requires additional scrutiny. This appendix provides a few recommendations on its architecture.

Simplify the Pre-Minting Functionality

The NFT contracts support the pre-minting of numerous tokens. These pre-minted tokens, which have no metadata, represent copies of original tokens and are used by the `Summoner` contract to transfer copies of tokens to users.

To simplify this architecture, implement one of the following recommendations:

- **Mint a new `MosaicNFT` token each time a new NFT is summoned.** This will simplify the logic around the creation of pre-minted tokens.
- **Require that pre-minted tokens be used to summon NFTs.** This will ensure that all NFTs are pre-minted before they are summoned.

Distinguish between the Types of Tokens Held by the Summoner

As highlighted in TOB-CMP-23, the `Summoner` contract's ability to hold multiple types of tokens makes it difficult to track NFT ownership.

Take one of the steps outlined below to simplify this architecture:

- **Lock and seal original tokens while the phantom copies are being minted and burned.** This will reduce blockchain bloat and simplify the off-chain process of differentiating between original NFTs and their copies.
- **Implement one or more additional data structures to keep track of NFT copies and originals.** Implementing mappings that store source / destination IDs, NFT copies, and token metadata, for example, would facilitate token tracking.

Use Only One ID for Each Token

The `Summoner` contract creates a unique ID for each token by hashing the chain's block number, the chain ID, the chain's contract address, and the contract's nonce. Each ID maps to a `MosaicNFT` ID containing a hash of the original NFT address, the ID of the origin network, and the NFT ID created by the `Summoner` contract.

We recommend using one ID for each token and mapping that ID to the token on every chain. This will facilitate the tracking of tokens that are bridged cross-chain.

F. Roles and Privileges

A survey of the roles controlled by single private keys and those roles' privileges is provided below.

Role	Privileges
`MosaicVault` (owner private key)	• Upgrading the contract • Changing the relayer's address • Updating the `MosaicVaultConfig` address • Granting / denying user withdrawals • Withdrawing contract funds • Minting / burning receipt tokens • Pausing / unpausing contracts • Transferring funds through `digestFunds`
`BridgeAggregator` (owner private key)	• Upgrading the contract • Setting the `MosaicVault` address • Adding / removing bridges
`MosaicVaultConfig` (owner private key)	• Changing contract addresses • Updating the supported automated market makers • Changing token ratios to rebalance liquidity • Adding tokens to / removing them from the `MosaicVault` whitelist • Adding tokens to / removing them from the Mosaic network whitelist • Controlling fee thresholds • Controlling block thresholds • Restricting the release of liquidity • Pausing / unpausing contracts
`MosaicHolding` (admin private key)	• Upgrading the contract • Adding investment strategies • Transferring funds • Creating new roles • Pausing / unpausing contracts • Rebalancing funds
Relayer (private key)	• Granting / denying user withdrawals

	• Transferring funds (ERC20 and ERC721 tokens) • Charging fees • Minting / burning receipt tokens
`MosaicNFT` (owner private key)	• Setting minter addresses
`Summoner` (owner private key)	• Upgrading the contract • Controlling fees • Updating the `MosaicNFT` address • Setting the relayer address • Pausing / unpausing contracts • Releasing seals on NFTs • Summoning NFTs across chains • Pre-minting Mosaic NFTs
`SummonerConfig` (owner private key)	• Setting the `transferLockupTime` value • Changing the fee tokens and fee amounts • Pausing / unpausing the network
`MsgReceiverFactory` (owner private key)	• Updating the contract • Adding fee tokens to / removing them from the whitelist • Setting the relayer address • Pausing / unpausing contracts • Removing `MsgReceiver` personas
`MsgReceiver` (owner private key)	• Updating the contract • Forwarding calls to other smart contracts • Approving tokens and calling other contracts • Transferring funds
`MsgSender` (owner private key)	• Updating the contract • Transferring funds • Adding networks to / removing them from the whitelist • Pausing / unpausing a network • Pausing / unpausing contracts
Rebalancing Bot	• Transferring funds to an EOA • Rebalancing funds