CompliFi

Security Assessment

July 6, 2021

Prepared For:
Dimitri Senchenko | *CompliFi*
ds@compli.fi

Prepared By:
Michael Colburn | *Trail of Bits*
michael.colburn@trailofbits.com

Maximilian Krüger | *Trail of Bits*
max.kruger@trailofbits.com

Devashish Tomar | *Trail of Bits*
devashish.tomar@trailofbits.com

Executive Summary
Project Dashboard
Code Maturity Evaluation
Engagement Goals
Coverage
Recommendations Summary
- Short term
- Long term

Findings Summary
- 1. Vault.changeState does not correctly emit the old state
- 2. Contracts used as dependencies do not track upstream changes
- 3. Initialization functions can be front-run
- 4. Lack of access modifiers on Vault.initialize leaves it susceptible to front-running
- 5. Lack of zero-value checks on functions
- 6. Solidity compiler optimizations can be problematic
- 7. Lack of contract and user documentation
- 8. Missing events for critical operations
- 9. Vault.constructor would benefit from an additional check of collateralSplit

A. Vulnerability Classifications
B. Code Maturity Classifications
C. Code Quality Recommendations
D. Token Integration Checklist
- General Security Considerations
- ERC Conformity
- Contract Composition
- Owner privileges
- Token Scarcity

Executive Summary

From June 14 to July 1, 2021, CompliFi engaged Trail of Bits to review the security of its derivative-issuance and AMM Solidity smart contracts. Trail of Bits conducted this assessment over six person-weeks, with three engineers working from commit hash `2bb3981` from the CompliFi/complifi-protocol-internal repository and commit hash `f6ec61a` from the CompliFi/complifi-amm-internal repository.

July 8, 2021 Update: After the conclusion of the assessment, CompliFi pushed the reviewed contracts to public repositories. The reviewed `contracts` subfolder of commit `2bb3981` in CompliFi/complifi-protocol-internal is identical to the `contracts` subfolder of commit `912e930` in CompliFi/complifi-protocol. The reviewed `contracts` subfolder of commit `f6ec61a` in CompliFi/complifi-amm-internal is identical to that of commit `827674f` in CompliFi/complifi-amm.

During the first week of the engagement, we reviewed the CompliFi system's documentation and began to assess the derivative-issuance protocol. Our efforts consisted of a manual review as well as an automated review using Slither, our Solidity static analyzer. In the second week, we focused on reviewing the AMM protocol. In the final week of the engagement, we conducted an in-depth end-to-end review of the entire system.

Trail of Bits identified nine issues ranging from medium to informational severity. The medium-severity issue and one low-severity issue involve certain contracts' initialization functions, which are vulnerable to front-running. The other low-severity issue involves third-party dependencies that are committed directly to the repository without any indication of their versions or modifications. The remaining issues, all of which are of informational severity, are related to missing or incorrect events, a lack of validation when setting addresses, a lack of documentation, and the use of optional compiler optimizations.

We also identified several code quality concerns not related to any particular security issues, which are discussed in Appendix C. Appendix D includes guidance on interacting with third-party tokens.

The CompliFi system is broken up into well-defined, logical pieces and generally adheres to Solidity development best practices. While the codebase has some high-level documentation, it lacks coverage of certain fine-grained details of its expected functionality. This lack of coverage, combined with the occasional use of very concise function and variable names, hinders the readability of the code. Going forward, we suggest that the CompliFi team resolve the issues detailed in this report and prioritize improving the codebase's documentation and testing, especially when developing new types of derivative contracts.

Project Dashboard

Application Summary

Name	CompliFi
Versions	`2bb3981` `f6ec61a`
Type	Solidity
Platform	EVM

Engagement Summary

Dates	June 14–July 1, 2021
Method	Full knowledge
Consultants Engaged	3
Level of Effort	6 person-weeks

Vulnerability Summary

Total High-Severity Issues	0	
Total Medium-Severity Issues	1	■
Total Low-Severity Issues	2	■ ■
Total Informational-Severity Issues	6	■ ■ ■ ■ ■ ■
Total Undetermined-Severity Issues	0	
Total	9	

Category Breakdown

Auditing and Logging	2	■ ■
Configuration	2	■ ■
Data Validation	2	■ ■
Documentation	1	■
Patching	1	■
Undefined Behavior	1	■
Total	9	

Code Maturity Evaluation

Category Name	Description
Access Controls	**Satisfactory.** There were appropriate access controls in place for privileged operations.
Arithmetic	**Satisfactory.** The code generally used safe math functions to perform calculations, and we did not identify any potential overflows in places in which these functions were not used. However, since some calculations at the core of the protocol's functionality are quite complex, the documentation and testing of the code should be more thorough.
Assembly Use/Low-Level Calls	**Satisfactory.** The use of assembly was limited, and appropriate return value checks were in place.
Centralization	**Satisfactory.** The owners of the `PoolFactory` and `VaultFactory` contracts can update the parameters of pools and vaults to be deployed in the future, but not existing ones. `Vault` owners can pause the `Vault`, blocking settlement and redemption operations, but cannot block token refunds. Pool contract owners can pause swapping operations, but liquidity can still be added or removed during such pauses.
Code Stability	**Strong.** The code did not change during the assessment.
Contract Upgradeability	**Satisfactory.** Several contracts used a standard OpenZeppelin proxy.
Function Composition	**Satisfactory.** The functions and contracts were organized and scoped appropriately.
Front-Running	**Moderate.** Certain initialization functions of the `FeeLoggerProxy`, `VaultFactory`, and `Vault` contracts were vulnerable to front-running.
Monitoring	**Moderate.** Several functions in the `VaultFactory` and `PoolFactory` contracts (specifically those that modify the system parameters of contracts to be deployed in the future) did not emit events.
Specification	**Moderate.** The code's high-level documentation should be supplemented by more accessible and comprehensive lower-level documentation. The comment coverage in the AMM codebase should also be improved.

Testing & Verification	**Moderate.** The system's functionality had adequate basic test coverage. However, this testing should be augmented by thorough unit tests and more rigorous testing of edge cases and expected extreme values.

Engagement Goals

The engagement was scoped to provide a security assessment of the CompliFi derivative-issuance and AMM smart contracts at commit hashes `2bb3981` and `f6ec61a` from the [CompliFi/complifi-protocol-internal]() repository and [CompliFi/complifi-amm-internal]() repository, respectively.

Specifically, we sought to answer the following questions:

- Is it possible for participants to steal or lose tokens?
- Are there appropriate access controls on the system components?
- Could participants perform denial-of-service or phishing attacks against any of the system components?
- Do the repricing algorithms arrive at the expected price?
- Are derivatives correctly priced upon their settlement?

Coverage

Vault. Vaults allow users to deposit the collateral needed to mint and later redeem derivative tokens. We reviewed these contracts to ensure that the vault lifecycle state transitions were sound, that accurate bookkeeping was performed throughout, and that the factory contracts properly constructed new vaults.

Pool. The `Pool` system is a modified version of the Balancer v1 AMM. We reviewed the contracts to ensure that any changes made after the fork would not introduce vulnerabilities, that the `Pool` correctly handled derivative tokens, and that the factory contracts properly constructed new pools.

Registries. The registry contracts serve as directories of the building blocks that can be used to construct a new derivative or pool. We reviewed these contracts to ensure that they accurately tracked the underlying components and were used in a sound manner.

Repricers. The derivative repricer contracts ensure that the AMM uses accurate prices for derivative tokens throughout their lifecycle. We reviewed these contracts to ensure that they returned accurate prices and that their calculations would not result in arithmetic overflows.

Recommendations Summary

This section aggregates all the recommendations made during the engagement. Short-term recommendations address the immediate causes of issues. Long-term recommendations pertain to the development process and long-term design goals.

Short Term

❑ **Either reorder the statements so that an event is emitted before the variable is updated or cache the old state in a temporary variable.** TOB-CMF-001

❑ **Review the codebase and document the source and version of each third-party dependency.** Consider including third-party sources as submodules in your Git repository to maintain internal path consistency and ensure that any dependencies are updated periodically. TOB-CMF-002

❑ **Ensure that the contract passes initialization data to the `TransparentUpgradeableProxy` constructor or that the deployment scripts have robust protections against front-running attacks.** TOB-CMF-003

❑ **Implement an access control modifier for the `initialize` function.** Develop clear documentation warning users that the values of derivatives may be unusual and reminding them to carefully check information on derivatives. TOB-CMF-004

❑ **Add zero-value checks on all function arguments to ensure users cannot accidentally set incorrect values, misconfiguring the system.** TOB-CMF-005

❑ **Measure the gas savings from optimizations and carefully weigh them against the possibility of an optimization-related bug.** TOB-CMF-006

❑ **Review and properly document the details currently missing from the documentation.** TOB-CMF-007

❑ **Add events for all critical operations.** Events aid in contract monitoring and the detection of suspicious behavior. TOB-CMF-008

❑ **Add `require(collateralSplit.isCollateralSplit())` to the `Vault` constructor.** TOB-CMF-009

Long Term

❑ **Consider adding tests for events to ensure they are emitted correctly.** TOB-CMF-001

❑ **Use an Ethereum development environment and NPM to manage packages in the project.** TOB-CMF-002

❑ **Carefully review the Solidity documentation, especially the "Warnings" section, as well as the pitfalls of using the `delegatecall` proxy pattern.** TOB-CMF-003

❑ **Analyze all contracts to identify any functions that could be front-run by attackers seeking to assign their own variables.** TOB-CMF-004

❑ **Use Slither, which will catch functions that do not have zero-value checks, and integrate it into the continuous integration pipeline.** TOB-CMF-005

❑ **Monitor the development and adoption of Solidity compiler optimizations to assess their maturity.** TOB-CMF-006

❑ **Consider writing a formal specification of the protocol.** TOB-CMF-007

❑ **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. TOB-CMF-008

❑ **Ensure that all input validation checks are as thorough as possible.** TOB-CMF-009

Findings Summary

#	Title	Type	Severity
1	Vault.changeState does not correctly emit the old state	Auditing and Logging	Informational
2	Contracts used as dependencies do not track upstream changes	Patching	Low
3	Initialization functions can be front-run	Configuration	Low
4	Lack of access modifiers on Vault.initialize leaves it susceptible to front-running	Configuration	Medium
5	Lack of zero-value checks on functions	Data Validation	Informational
6	Solidity compiler optimizations can be problematic	Undefined Behavior	Informational
7	Lack of contract and user documentation	Documentation	Informational
8	Missing events for critical operations	Auditing and Logging	Informational
9	Vault.constructor would benefit from an additional check of collateralSplit	Data Validation	Informational

1. Vault.changeState does not correctly emit the old state

Severity: Informational
Difficulty: Low
Type: Auditing and Logging
Finding ID: TOB-CMF-001
Target: `complifi-protocol-internal/contracts/Vault.sol`

Description

The `Vault` contract progresses through several states. Each time the contract transitions to a new state, the `changeState` function emits an event logging both the previous and new states.

```
enum State { Created, Live, Settled }

event StateChanged(State oldState, State newState);
```

Figure 1.1: `contracts/Vault.sol#L39-31`

However, since the `changeState` function updates the `state` variable before emitting an event, the previous state is lost, and the new state is logged twice.

```
function changeState(State _newState) internal {
    state = _newState;
    emit StateChanged(state, _newState);
}
```

Figure 1.2: `contracts/Vault.sol#L39-31`

Exploit Scenario

Alice develops an application that interfaces with the CompliFi contracts and relies on events to track state changes in the protocol. Because the previous vault state is not properly logged, Alice's application provides inaccurate vault state data to its users.

Recommendations

Short term, either reorder the statements so that an event is emitted before the variable is updated or cache the old state in a temporary variable.

Long term, consider adding tests for events to ensure they are emitted correctly.

2. Contracts used as dependencies do not track upstream changes

Severity: Low
Difficulty: Low
Type: Patching
Finding ID: TOB-CMF-002
Target: `*/contracts/libs`

Description
`BokkyPooBahsDateTimeLibrary` and several OpenZeppelin contracts have been copied and pasted into the repositories under review. The code documentation does not specify the exact revision that was made or whether it was modified. As such, the contracts may not reliably reflect updates or security fixes implemented in their dependencies, as those changes must be manually integrated into the contracts.

Exploit Scenario
A third-party contract used in date-time calculations receives an update with a critical fix for a vulnerability. An attacker detects the use of a vulnerable contract and exploits the vulnerability against one of the contracts.

Recommendations
Short term, review the codebase and document the source and version of each third-party dependency. Consider including third-party sources as submodules in your Git repository to maintain internal path consistency and ensure that any dependencies are updated periodically.

Long term, use an Ethereum development environment and NPM to manage packages in the project.

3. Initialization functions can be front-run

Severity: Low
Difficulty: High
Type: Configuration
Finding ID: TOB-CMF-003
Target: `contracts/{FeeLoggerProxy, VaultFactory}.sol`

Description

July 6, 2021 Update: The severity of this issue was initially set to medium. However, we agreed to lower the rating after CompliFi provided the following additional context: "The attack is always detectable, since the transaction that is being front-run will revert, so the tampered version of the protocol would simply be discarded and all we would need to do is redeploy." Since the protocol is deployed only once and CompliFi could detect a front-running attack, discard the deployment attempt, and launch a second attempt, a front-running attack would likely have no impact beyond obstructing the deployment process.

The `FeeLoggerProxy` and `VaultFactory` implementation contracts have initialization 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 a constructor, and they have initializer functions:

```
/// @notice Initializes vault factory contract storage
/// @dev Used only once when vault factory is created for the first time
function initialize(
    address _derivativeSpecificationRegistry,
    address _oracleRegistry,
    address _oracleIteratorRegistry,
    address _collateralTokenRegistry,
    address _collateralSplitRegistry,
    address _tokenBuilder,
    address _feeLogger,
    uint256 _protocolFee,
    address _feeWallet,
    uint256 _authorFeeLimit,
    address _vaultBuilder,
    uint256 _settlementDelay
) external initializer {
    __Ownable_init();

    setDerivativeSpecificationRegistry(_derivativeSpecificationRegistry);
    setOracleRegistry(_oracleRegistry);
    setOracleIteratorRegistry(_oracleIteratorRegistry);
```

```
        setCollateralTokenRegistry(_collateralTokenRegistry);
        setCollateralSplitRegistry(_collateralSplitRegistry);
        ...
```

Figure 3.1: `complifi-protocol-internal/contracts/VaultFactory.sol#L43-L65`

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

Exploit Scenario

Bob deploys the `VaultFactoryProxy` contract, which passes empty data to the parent constructor, bypassing the built-in initialization trigger. Bob must execute a separate transaction to invoke the `initialize` function and finish setting up the `VaultFactory`. Eve front-runs the contract's initialization and sets a registry under her control as the `_oracleRegistry`. If this attack remains undetected, Eve will be able to manipulate the price feed that Bob's vaults will rely on once deployed.

Recommendations

Short term, ensure that the contract passes initialization data to the `TransparentUpgradeableProxy` constructor or that the deployment scripts have robust protections against front-running attacks.

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

4. Lack of access modifiers on Vault.initialize leaves it susceptible to front-running

Severity: Medium
Difficulty: High
Type: Configuration
Finding ID: TOB-CMF-004
Target: `Vault.sol`

Description

July 9, 2021 Update: The severity of this issue was initially set to high. However, we agreed to lower the severity rating after CompliFi provided the following additional context:

1. *"Anyone is free to call Vault.initialize, and in practice it is the person who wants to create a particular derivative for whatever purpose."*
2. *"Minting of derivative is a risk-free event at all parametrisation levels - the user receives equal amounts of long and short positions."*
3. *"If the attacker manages to alter the intended parametrisation of a derivative, the AMM would not allow them to dispose of it at anything other than fair market value."*

The `Vault` implementation contract has an initialization function that can be front-run, allowing an attacker to incorrectly initialize the contract.

The `Vault` contract is initialized through a two-step process, first through its constructor and then through the `initialize` function.

```
/// @notice Initialize vault by creating derivative token and switching to Live state
/// @dev Extracted from constructor to reduce contract gas creation amount
function initialize(int256[] calldata _underlyingStarts) external {
    require(state == State.Created, "Incorrect state.");

    underlyingStarts = _underlyingStarts;

    changeState(State.Live);

    (primaryToken, complementToken) = tokenBuilder.buildTokens(
        derivativeSpecification,
        settleTime,
        address(collateralToken)
    );

    emit LiveStateSet(address(primaryToken), address(complementToken));
}
```

Figure 4.1: `complifi-protocol-internal/contracts/Vault.sol#L172-L188`

An attacker could front-run this function and initialize instances of the `Vault` contract with a malicious `_underlyingStarts` value.

Exploit Scenario
Bob deploys the `Vault` contract through the `VaultFactory` and must execute a separate transaction to invoke `initialize` and finish setting up the `Vault`. Eve front-runs the contract's initialization and sets an arbitrary value as `_underlyingStarts`. This leads to the creation of a valid but unusual derivative. Certain users fail to carefully read the derivative and make incorrect assumptions about its properties, which could lead to a loss of funds.

Recommendations
Short term, implement an access control modifier for the `initialize` function. Develop clear documentation warning users that the values of derivatives may be unusual and reminding them to carefully check information on derivatives.

Long term, analyze all contracts to identify any functions that could be front-run by attackers seeking to assign their own variables.

5. Lack of zero-value checks on functions

Severity: Informational
Difficulty: High
Type: Data Validation
Finding ID: TOB-CMF-005
Target: `complifi-protocol-internal/contracts/DerivativeSpecification.sol`

Description
Certain setter functions fail to validate incoming arguments, so callers can accidentally set important state variables to the zero address.

The constructor in the derivative-issuance protocol's `DerivativeSpecification` contract takes as a parameter the address of the specification's author, who then earns fees on the specification:

```
constructor(
    address _author,
    string memory _name,
    string memory _symbol,
    bytes32[] memory _oracleSymbols,
    bytes32[] memory _oracleIteratorSymbols,
    bytes32 _collateralTokenSymbol,
    bytes32 _collateralSplitSymbol,
    uint256 _livePeriod,
    uint256 _primaryNominalValue,
    uint256 _complementNominalValue,
    uint256 _authorFee,
    string memory _baseURI
) public {
    author_ = _author;
    ...
}
```

Figure 5.1: `contracts/DerivativeSpecification.sol#L115-L129`

When a zero address is provided, the `Vault` of this derivative will attempt to credit fees to the zero address. As a result, depending on whether the collateral token allows transfers to the zero address, the fees will be burned and lost forever, or attempts to `mint` derivatives from the `Vault` will revert.

Exploit Scenario
Alice, a derivative specification author, registers her specification with the CompliFi protocol but accidentally provides `address(0)` as the author address. A vault that uses her specification is created. Bob deposits collateral into the vault and mints derivative tokens

from it. The amount of Alice's fee is deducted from Bob's deposit, but the fee is transferred to `address(0)` and lost forever.

Recommendations

Short term, add zero-value checks on all function arguments to ensure users cannot accidentally set incorrect values, misconfiguring the system.

Long term, use Slither, which will catch functions that do not have zero-value checks, and integrate it into the continuous integration pipeline.

6. Solidity compiler optimizations can be problematic

Severity: Informational
Type: Undefined Behavior
Target: `truffle-config.js`, `hardhat.config.ts`

Difficulty: Low
Finding ID: TOB-CMF-006

Description
The `complifi-protocol-internal` and `complifi-amm-internal` repositories 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 CompliFi smart 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.

7. Lack of contract and user documentation

Severity: Informational
Difficulty: Low
Type: Documentation
Finding ID: TOB-CMF-007
Target: `Throughout`

Description
Parts of the codebase lack code documentation, high-level descriptions, and examples, making the contracts difficult to review and increasing the likelihood of developer and user mistakes.

The documentation would benefit from the following details:

- The formulas used for the repricing algorithms
- An explanation of dynamic fee calculation logic
- NatSpec comments on all AMM contracts

Where relevant, the documentation on each of these items should include its expected properties and assumptions.

Recommendations
Short term, review and properly document the items mentioned above.

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

8. Missing events for critical operations

Severity: Informational
Difficulty: Low
Type: Auditing and Logging
Finding ID: TOB-CMF-008
Target: `Throughout`

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.

The following critical operations would benefit from triggering events:

- `VaultFactory`
  - `setProtocolFee`
  - `setAuthorFeeLimit`
  - `setTokenBuilder`
  - `setFeeLogger`
  - `setVaultBuilder`
  - `setSettlementDelay`
  - `setDerivativeSpecificationRegistry`
  - `setOracleRegistry`
  - `setOracleIteratorRegistry`
  - `setCollateralTokenRegistry`
  - `setCollateralSplitRegistry`
- `PoolFactory`
  - `setPoolBuilder`
  - `setDynamicFee`
  - `setRepricerRegistry`

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

Exploit Scenario
Eve compromises the governance address and, by calling `setProtocolFee`, changes the vault protocol fee to the maximum amount. Because no events are emitted, Bob does not notice the compromise when depositing assets into new vaults. Alice is then able to collect much higher fees than she otherwise could until the change is detected.

Recommendations
Short term, add events for all critical operations. 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.

9. Vault.constructor would benefit from an additional check of collateralSplit

Severity: Informational
Difficulty: High
Type: Data Validation
Finding ID: TOB-CMF-009
Target: `Vault.sol`

Description
The `CollateralSplit` contract's `isCollateralSplit` function can be reused in the `Vault`'s constructor to further ensure the integrity of data.

Exploit Scenario
An address that does not implement `ICollateralSplit` is passed as an argument to the `Vault.constructor`. As a result, calls to `settle` will always revert, and assets will remain frozen in the `Vault`.

Recommendations
Short term, add `require(collateralSplit.isCollateralSplit())` to the `Vault` constructor.

Long term, ensure that all input validation checks are as thorough as possible.

A. Vulnerability Classifications

Vulnerability Classes	
Class	**Description**
Access Controls	Related to authorization of users and assessment of rights
Auditing and Logging	Related to auditing of actions or logging of problems
Authentication	Related to the identification of users
Configuration	Related to security configurations of servers, devices, or software
Cryptography	Related to protecting the privacy or integrity of data
Data Exposure	Related to unintended exposure of sensitive information
Data Validation	Related to improper reliance on the structure or values of data
Denial of Service	Related to causing a system failure
Error Reporting	Related to the reporting of error conditions in a secure fashion
Patching	Related to keeping software up to date
Session Management	Related to the identification of authenticated users
Testing	Related to test methodology or test coverage
Timing	Related to race conditions, locking, or the order of operations
Undefined Behavior	Related to undefined behavior triggered by the program

Severity Categories	
Severity	**Description**
Informational	The issue does not pose an immediate risk but is relevant to security best practices or Defense in Depth.
Undetermined	The extent of the risk was not determined during this engagement.
Low	The risk is relatively small or is not a risk the customer has indicated is important.

Medium	Individual users' information is at risk; exploitation could pose reputational, legal, or moderate financial risks to the client.
High	The issue could affect numerous users and have serious reputational, legal, or financial implications for the client.

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploitation was not determined during this engagement.
Low	The flaw is commonly exploited; public tools for its exploitation exist or can be scripted.
Medium	An attacker must write an exploit or will need in-depth knowledge of a complex system.
High	An attacker must have privileged insider access to the system, may need to know extremely complex technical details, or must discover other weaknesses to exploit this issue.

B. Code Maturity Classifications

Code Maturity Classes	
Category Name	**Description**
Access Controls	Related to the authentication and authorization of components
Arithmetic	Related to the proper use of mathematical operations and semantics
Assembly Use	Related to the use of inline assembly
Centralization	Related to the existence of a single point of failure
Upgradeability	Related to contract upgradeability
Function Composition	Related to separation of the logic into functions with clear purposes
Front-Running	Related to resilience against front-running
Key Management	Related to the existence of proper procedures for key generation, distribution, and access
Monitoring	Related to the use of events and monitoring procedures
Specification	Related to the expected codebase documentation
Testing & Verification	Related to the use of testing techniques (unit tests, fuzzing, symbolic execution, etc.)

Rating Criteria	
Rating	**Description**
Strong	The component was reviewed, and no concerns were found.
Satisfactory	The component had only minor issues.
Moderate	The component had some issues.
Weak	The component led to multiple issues; more issues might be present.
Missing	The component was missing.

Not Applicable	The component is not applicable.
Not Considered	The component was not reviewed.
Further Investigation Required	The component requires further investigation.

C. Code Quality Recommendations

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

General Recommendation

- **Avoid using overly concise or abbreviated names for functions and variables.** More informative names will greatly enhance the readability of code that requires domain-specific knowledge or is called from other contracts.

CollateralSplitParent.sol

- **The `split` function takes an array of oracles and asserts that the array contains exactly one oracle.** Passing in the oracle as a single address variable would reduce the code's complexity while retaining its functionality.

ICollateralSplitTemplate.sol

- **`ICollateralSplitTemplate` is not used anywhere in the codebase and appears to be dead code.** Consider removing it to reduce the complexity of the codebase.

Vault.sol

- **The `range` function is used only once, in line 219, `split = range(split)`, which is not self-explanatory.** To reduce the amount of indirection, consider removing the `range` function and making line 219 more self-explanatory by changing it to the following: `split = max(FRACTION_MULTIPLIER, split)`.

VaultFactory.sol

- **The `setDerivativeSpecification`, `setOracle`, `setOracleIterator`, `setCollateralToken`, and `setCollateralSplit` functions start with "`set`" but differ from the other setters in that they add addresses to the registries instead of setting state variables.** Consider prefixing them with "`register`" instead of "`set`" to distinguish them from the other setters.

Ownable.sol

- **The `Ownable.sol` file contains a contract named `OwnableUpgradeSafe`.** To clarify the content of the file, consider renaming it `OwnableUpgradeSafe.sol`.

D. Token Integration Checklist

The following checklist provides recommendations for interactions with arbitrary tokens. Every unchecked item should be justified, and its associated risks, understood. An up-to-date version of the checklist can be found in crytic/building-secure-contracts.

For convenience, all Slither utilities can be run directly on a token address, such as the following:

```
slither-check-erc 0xdac17f958d2ee523a2206206994597c13d831ec7 TetherToken
```

To follow this checklist, use the below output from Slither for the token:

```
- slither-check-erc [target] [contractName] [optional: --erc ERC_NUMBER]
- slither [target] --print human-summary
- slither [target] --print contract-summary
- slither-prop . --contract ContractName # requires configuration, and use of Echidna and Manticore
```

General Security Considerations

- ❏ **The contract has a security review.** Avoid interacting with contracts that lack a security review. Check the length of the assessment (i.e., the level of effort), the reputation of the security firm, and the number and severity of the findings.
- ❏ **You have contacted the developers.** You may need to alert their team to an incident. Look for appropriate contacts on blockchain-security-contacts.
- ❏ **They have a security mailing list for critical announcements.** Their team should advise users (like you!) when critical issues are found or when upgrades occur.

ERC Conformity

Slither includes a utility, `slither-check-erc`, that reviews the conformance of a token to many related ERC standards. Use slither-check-erc to review the following:

- ❏ **`Transfer` and `transferFrom` return a boolean.** Several tokens do not return a boolean on these functions. As a result, their calls in the contract might fail.
- ❏ **The `name`, `decimals`, and `symbol` functions are present if used.** These functions are optional in the ERC20 standard and may not be present.
- ❏ **`Decimals` returns a `uint8`.** Several tokens incorrectly return a `uint256`. In such cases, ensure that the value returned is below 255.
- ❏ **The token mitigates the known ERC20 race condition.** The ERC20 standard has a

known ERC20 race condition that must be mitigated to prevent attackers from stealing tokens.
- ❏ **The token is not an ERC777 token and has no external function call in `transfer` or `transferFrom`.** External calls in the transfer functions can lead to reentrancies.

Slither includes a utility, [`slither-prop`](), that generates unit tests and security properties that can discover many common ERC flaws. Use slither-prop to review the following:

- ❏ **The contract passes all unit tests and security properties from `slither-prop`.** Run the generated unit tests and then check the properties with [Echidna]() and [Manticore]().

Finally, there are certain characteristics that are difficult to identify automatically. Conduct a manual review of the following conditions:

- ❏ **`Transfer` and `transferFrom` should not take a fee.** Deflationary tokens can lead to unexpected behavior.
- ❏ **Potential interest earned from the token is taken into account.** Some tokens distribute interest to token holders. This interest may be trapped in the contract if not taken into account.

Contract Composition

- ❏ **The contract avoids unnecessary complexity.** The token should be a simple contract; a token with complex code requires a higher standard of review. Use Slither's [`human-summary`]() printer to identify complex code.
- ❏ **The contract uses `SafeMath`.** Contracts that do not use `SafeMath` require a higher standard of review. Inspect the contract by hand for `SafeMath` usage.
- ❏ **The contract has only a few non-token-related functions.** Non-token-related functions increase the likelihood of an issue in the contract. Use Slither's [contract-summary]() printer to broadly review the code used in the contract.
- ❏ **The token has only one address.** Tokens with multiple entry points for balance updates can break internal bookkeeping based on the address (e.g., balances[token_address][msg.sender] may not reflect the actual balance).

Owner Privileges

- ❏ **The token is not upgradeable.** Upgradeable contracts may change their rules over time. Use Slither's [`human-summary`]() printer to determine if the contract is upgradeable.
- ❏ **The owner has limited minting capabilities.** Malicious or compromised owners can abuse minting capabilities. Use Slither's [`human-summary`]() printer to review minting capabilities, and consider manually reviewing the code.

- ❏ **The token is not pausable.** Malicious or compromised owners can trap contracts relying on pausable tokens. Identify pausable code by hand.
- ❏ **The owner cannot blacklist the contract.** Malicious or compromised owners can trap contracts relying on tokens with a blacklist. Identify blacklisting features by hand.
- ❏ **The team behind the token is known and can be held responsible for abuse.** Contracts with anonymous development teams or teams that reside in legal shelters require a higher standard of review.

Token Scarcity

Reviews of token scarcity issues must be executed manually. Check for the following conditions:

- ❏ **The supply is owned by more than a few users.** If a few users own most of the tokens, they can influence operations based on the tokens' repartition.
- ❏ **The total supply is sufficient.** Tokens with a low total supply can be easily manipulated.
- ❏ **The tokens are located in more than a few exchanges.** If all the tokens are in one exchange, a compromise of the exchange could compromise the contract relying on the token.
- ❏ **Users understand the risks associated with a large amount of funds or flash loans.** Contracts relying on the token balance must account for attackers with a large amount of funds or attacks executed through flash loans.
- ❏ **The token does not allow flash minting.** Flash minting can lead to substantial swings in the balance and the total supply, which necessitate strict and comprehensive overflow checks in the operation of the token.