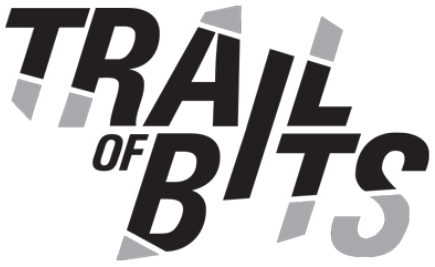

Beta Finance

Security Assessment

August 23, 2021

Prepared For:
Allen Lee | *Beta Finance*
allen@betafinance.org

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

Alexander Remie | *Trail of Bits*
alexander.remie@trailofbits.com

Scott Sunarto | *Trail of Bits*
scott.sunarto@trailofbits.com

Simone Monica | *Trail of Bits*
simone.monica@trailofbits.com

Executive Summary

From August 9 to August 20, 2021, Beta Finance engaged Trail of Bits to review the security of the Beta Finance smart contracts. Trail of Bits conducted this assessment over four person-weeks, with two engineers working from commit hash `f12ca73` from the `beta-finance/beta` repository.

During the first week of the assessment, we familiarized ourselves with the codebase and looked for common Solidity issues. We also performed an automated review using Slither, our Solidity static analyzer. In the final week of the assessment, we performed an in-depth review of the entire system and its interactions with external contracts.

We identified nine issues ranging from high to informational severity. The high-severity issue and the medium-severity issue concern the recovery of tokens sent to `BToken` contracts. The low-severity issue involves several contracts and functions that would benefit from additional address input validation. One informational-severity issue concerns the lack of fixes for issues that were reported in previous audits. The remaining informational-severity issues relate to the passing of the token `symbol` rather than the token `name` to parent constructors, the use of the EIP-2612 `permit` function and risky Solidity compiler optimizations, the reliance on `tx.origin` to prevent contracts from interacting with the system, and the `recover` function's failure to emit an event.

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

The Beta Finance contracts generally follow smart contract development best practices. Functions and contracts are scoped and grouped appropriately, though some variable names are terse (e.g., `pid`). While most contracts have thorough comments, the `BetaRunner` contracts, given the complexity of the operations they execute, need much more thorough comment coverage. Going forward, we suggest that the Beta Finance team resolve the issues reported in this audit, improve the developer-level documentation, and provide robust user-level documentation, especially on the capabilities of privileged contract roles and any safeguards that may prevent their misuse.

Project Dashboard

Application Summary

Name	Beta Finance
Version	`f12ca73`
Type	Solidity
Platform	EVM

Engagement Summary

Dates	August 9–August 20, 2021
Method	Full knowledge
Consultants Engaged	2
Level of Effort	4 person-weeks

Vulnerability Summary

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

Category Breakdown

Access Controls	1	■
Auditing and Logging	1	■
Configuration	1	■
Data Validation	4	■■■■
Patching	1	■
Undefined Behavior	1	■
Total	9	

Code Maturity Evaluation

Category Name	Description
Access Controls	**Satisfactory.** There are appropriate access controls in place for privileged operations.
Arithmetic	**Satisfactory.** The contracts use a version of the Solidity compiler that includes built-in overflow checks.
Assembly Use/Low-Level Calls	**Satisfactory.** The contracts use assembly code only through the `BytesLib` library. All low-level calls are properly checked for success.
Centralization	**Weak.** The `recover` function is intended to allow the recovery of accidentally deposited tokens, but the `BetaBank governor` has the ability to recover any token held by the individual `BToken` contracts.
Code Stability	**Satisfactory.** The commit under review was updated once, on the second day of the assessment.
Contract Upgradeability	**Satisfactory.** The Beta Finance system uses a standard OpenZeppelin upgradeable proxy pattern to allow the `BetaBank` contract to be upgraded.
Function Composition	**Satisfactory.** The functions and contracts are organized and scoped appropriately.
Front-Running	**Satisfactory.** We did not identify any issues related to front-running.
Monitoring	**Moderate.** Adequate events are emitted for all functions but the `recover` function in the `BToken` contract. Given the power of this function, it must emit an event so that recovery operations will be evident and that users will be able to trust the system.
Specification	**Moderate.** With the exception of the `BetaRunner` contracts, most contracts have adequate comment coverage. However, we were not provided with any developer or user documentation.
Testing & Verification	**Satisfactory.** The repository includes a suite of tests for a variety of scenarios.

Engagement Goals

The engagement was scoped to provide a security assessment of the Beta Finance Solidity smart contracts in the `beta-finance/beta` repository at commit hash `f12ca73`.

Specifically, we sought to answer the following questions:

- Is it possible for an attacker to manipulate prices?
- Is the arithmetic for calculations and bookkeeping sound?
- Are there any reentrancy vulnerabilities in the contracts?
- Is the system susceptible to attacks such as flash loan attacks?

Coverage

`BetaBank`. The `BetaBank` contract is the core of the Beta Finance protocol. It keeps track of user positions and the available markets and facilitates interactions between them. We reviewed the contract to ensure that adequate access controls are in place, positions are tracked properly, and interactions between system components are sound.

`BToken`. `BToken` contracts enable the lending and borrowing of assets in the Beta Finance system. We reviewed the contract to ensure the accuracy of borrowing and repayment bookkeeping and interest accrual.

`BetaOracleUniswapV2`. This contract fetches raw prices from Uniswap V2 and calculates the pricing information for the Beta Finance contracts. We reviewed the contract to ensure that pricing information is calculated correctly and is used properly by the other Beta Finance contracts.

`BetaRunner`. The `BetaRunner` contracts are helper contracts that facilitate interactions between the Beta Finance system and external systems such as Uniswap. We reviewed the contracts to identify logic errors and to ensure that interactions with external systems are sound.

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

❑ **Add basic input validation mechanisms to the functions identified in the finding.** For example, add a check for the `0x0` address, which is the default value for an uninitialized address. TOB-BFI-001

❑ **Fix all known issues and review each fix.** TOB-BFI-002

❑ **Update the `BetaToken` constructor to pass the same `name` value to both parent contracts.** TOB-BFI-003

❑ **Add a `require(_token != underlying)` statement to `BToken`'s `recover` function.** TOB-BFI-004

❑ **Add the missing event emission to `BetaBank.recover`.** TOB-BFI-005

❑ **Consider the impacts of using the `onlyEOA` modifier, which may become ineffective in preventing contracts from interacting with the Beta Finance system.** TOB-BFI-006

❑ **Use the Uniswap `swap` function's callback arguments to calculate the exact collateral amount to send to users when they close short positions.** TOB-BFI-007

❑ **Measure the gas savings from optimizations and carefully weigh them against the possibility of an optimization-related bug.** TOB-BFI-008

❑ **Document the risk of `permit` front-running and ensure that external contracts and scripts reflect this possibility.** Alternatively, develop user documentation and on-chain mitigations to reduce the likelihood of a successful phishing campaign. TOB-BFI-009

Long term

❑ **Ensure that all functions perform some type of input validation and use Slither, which detects missing `0x0` address checks.** TOB-BFI-001

❑ **Ensure that the schedule of security audits includes sufficient time for the team to implement and review fixes before audits begin.** TOB-BFI-002

❑ **Follow the conventions posed by common EIP implementations when using them.** TOB-BFI-003

❑ **Build safeguards into recovery flows to prevent them from becoming abuse vectors.** TOB-BFI-004

❑ **Ensure that events are emitted for all critical operations and consider using a blockchain-monitoring system to track suspicious behavior in the contracts.** The Beta Finance system relies on several contracts to behave as expected. A monitoring mechanism for critical events would quickly detect any compromised system components. TOB-BFI-005

❑ **Design contracts in such a way that they function properly regardless of the type of address used to invoke the contracts.** TOB-BFI-006

❑ **Write properties and test them using Echidna.** TOB-BFI-007

❑ **Monitor the development and adoption of Solidity compiler optimizations to assess their maturity.** TOB-BFI-008

❑ **Document best practices for users of the Beta Finance contracts.** In addition to taking other precautions, users must be extremely careful when signing messages, avoid signing messages from suspicious sources, and always require hashing schemes to be public. TOB-BFI-009

Findings Summary

#	Title	Type	Severity
1	Missing zero-address checks in constructor and configuration functions	Data Validation	Low
2	Known issues in the codebase have not been fixed	Patching	Informational
3	Token constructor passes ERC20 symbol value as ERC20Permit name	Data Validation	Informational
4	BToken allows the BetaBank governor to recover underlying tokens	Data Validation	High
5	The recover function does not emit an event	Auditing and Logging	Informational
6	Reliance on tx.origin to enforce EOA-only callers	Access Controls	Informational
7	Incorrectly sent tokens can be recovered by users	Data Validation	Medium
8	Solidity compiler optimizations can be problematic	Undefined Behavior	Informational
9	Risks associated with EIP-2612	Configuration	Informational

1. Missing zero-address checks in constructor and configuration functions

Severity: Low
Type: Data Validation
Target: `Several contracts`

Difficulty: High
Finding ID: TOB-BFI-001

Description
Because the constructors and configuration functions lack zero-address checks, contract addresses can be set incorrectly. Fixing an incorrect address would likely require a contract upgrade or redeployment.

```
constructor(address _betaBank, address _weth) {
  address bweth = IBetaBank(_betaBank).bTokens(_weth);
  require(bweth != address(0), 'BetaRunnerBase/no-bweth');
  IERC20(_weth).safeApprove(_betaBank, type(uint).max);
  IERC20(_weth).safeApprove(bweth, type(uint).max);
  betaBank = _betaBank;
  weth = _weth;
}
```

Figure 1.1: beta-contract/contracts/BetaRunnerBase.sol#L23-L30

If a contract's address is set to the zero address, calls to functions in that contract will revert. The addresses set in the constructor function cannot be changed through configuration functions; therefore, to remediate an incorrectly set address, one would have to redeploy the contract and any other contracts that contain non-modifiable references to it.

The `setPendingGovernor` function is also missing zero-address checks. This function can be called multiple times, so one can fix an incorrect address by simply calling the function again with the correct address. Although this problem can be remediated more easily, adding a zero-address check would be beneficial.

```
function setPendingGovernor(address _pendingGovernor) external onlyGov {
  pendingGovernor = _pendingGovernor;
  emit SetPendingGovernor(_pendingGovernor);
}
```

Figure 1.2: beta-contract/contracts/BetaBank.sol#L130-L133

The following functions are missing zero-address checks:

- `BetaRunnerBase.constructor`
- `BetaRunnerUniswapV2.constructor`
- `BetaRunnerUniswapV3.constructor`
- `BetaConfig.setPendingGovernor`
- `BetaBank.setPendingGovernor`
- `WETHGateway.burn`

- `BetaOracleUniswapV2.setPendingGovernor`

Exploit Scenario
An error or typo in the deployment script causes the contract address of a system component to be recorded as `0x0` in each of the system's contracts. As a result, the entire system must be redeployed or upgraded at a significant gas cost.

Recommendations
Short term, add basic input validation mechanisms to the functions identified in the finding. For example, add a check for the `0x0` address, which is the default value for an uninitialized address.

Long term, ensure that all functions perform some type of input validation and use Slither, which detects missing `0x0` address checks.

2. Known issues in the codebase have not been fixed

Severity: Informational
Type: Patching
Target: `Throughout`

Difficulty: High
Finding ID: TOB-BFI-002

Description
Several issues that the Beta Finance team is aware of have not been fixed. For example, the use of magic numbers instead of declared constants hampers readability and makes the code more difficult to maintain. This issue is still present. Unfixed known issues in the codebase increase the risks of compromise and make code reviews harder to conduct. Additionally, it is unclear whether the fixes will be correctly written before deployment.

Exploit Scenario
Eve is a malicious owner. She advertises that the codebase went through multiple audits, knowing that no fixes were applied. Alice and Bob start using the system. In modifying the interest rate calculation, Eve uses the wrong magic numbers, introducing an error. As a result of the bug, when Alice borrows from the system, she ends up paying significantly higher interest rates.

Recommendations
Short term, fix all known issues and review each fix.

Long term, ensure that the schedule of security audits includes sufficient time for the team to implement and review fixes before audits begin.

3. Token constructor passes ERC20 symbol value as ERC20Permit name

Severity: Informational
Type: Data Validation
Target: `beta-contract/contracts/BetaToken.sol`

Difficulty: Low
Finding ID: TOB-BFI-003

Description
The `BetaToken` contract is an ERC20 token that supports EIP-712-signed approvals. The constructor initializes the `name` of the token to `Beta Token` and the `symbol` to `BETA`.

```
    contract BetaToken is ERC20PresetMinterPauser('Beta Token', 'BETA'), ERC20Permit('BETA') {
```

Figure 3.1: `beta-contract/contracts/BetaToken.sol#L7`

The `ERC20Permit` contract's code comments recommend passing the `name` of the token to the EIP-712 domain separator. However, the `BetaToken` constructor passes the token's `symbol` value (`BETA`) rather than the `name` value (`Beta Token`) to the EIP-712 domain separator.

```
    /**
     * @dev Initializes the {EIP712} domain separator using the `name` parameter, and setting
`version` to `"1"`.
     *
     * It's a good idea to use the same `name` that is defined as the ERC20 token name.
     */
    constructor(string memory name) EIP712(name, "1") {}
```

Figure 3.2: `draft-ERC20Permit.sol#L30-L35` from OpenZeppelin's contract library

Exploit Scenario
Alice builds a dApp that interacts with the `BetaToken` contract and makes use of EIP-712-signed approvals. When her dApp generates the messages to be signed, it uses the token's `symbol` value to build the domain separator. As a result, the wrong domain separator is generated, and Alice's dApp fails to integrate properly with the contract.

Recommendations
Short term, update the `BetaToken` constructor to pass the same `name` value to both parent contracts.

Long term, follow the conventions posed by common EIP implementations when using them.

4. BToken allows the BetaBank governor to recover underlying tokens

Severity: High | Difficulty: High
Type: Data Validation | Finding ID: TOB-BFI-004
Target: `beta-contract/contracts/BetaBank.sol`, `beta-contract/contracts/BToken.sol`

Description
The `BToken` contract is an ERC20 token that supports the recovery of lost tokens using the `recover` function.

```
function recover(
  address _token,
  address _to,
  uint _amount
) external nonReentrant {
  require(msg.sender == betaBank, 'recover/not-BetaBank');
  if (_amount == type(uint).max) {
    _amount = IERC20(_token).balanceOf(address(this));
  }
  IERC20(_token).safeTransfer(_to, _amount);
}
```

Figure 4.1: `BToken.sol#L203-L214`

However, the `recover` function does not check whether the `_token` value is `underlying`. Without this check, `BetaBank`'s `governor` can recover all of the underlying tokens. As this attack would require a compromise of the `governor`, to further reduce the likelihood of this being exploited by a malicious insider or other attacker, the `governor` should be set to a multisig wallet or some type of decentralized governance.

```
function recover(
  address _bToken,
  address _token,
  uint _amount
) external onlyGov lock {
  require(underlyings[_bToken] != address(0), 'recover/not-bToken');
  BToken(_bToken).recover(_token, msg.sender, _amount);
}
```

Figure 4.2: `BetaBank.sol#L434-L441`

Exploit Scenario
Eve compromises the private key of the `BetaBank.governor`. She calls `recover` on all existing `BTokens` to transfer all of the underlying tokens to her account.

Recommendations
Short term, add a `require(_token != underlying)` statement to `BToken`'s `recover` function.

Long term, build safeguards into recovery flows to prevent them from becoming abuse vectors.

5. The recover function does not emit an event

Severity: Informational
Type: Auditing and Logging
Target: `beta-contract/contracts/BetaBank.sol`

Difficulty: Low
Finding ID: TOB-BFI-005

Description
Events emitted during contract execution aid in detection of suspicious activity, baselining of behavior, and monitoring. Without events, users and blockchain-monitoring systems cannot easily detect malfunctioning contracts, attacks, or behavior that falls outside of baseline conditions.

The `BetaBank.recover` function does not emit an event. Due to the custodial nature of the `BToken` contract, it must emit events during recovery operations to ensure proper recordkeeping and increased user awareness of these operations.

Exploit Scenario
An attacker discovers a vulnerability in the `BetaBank` contract and is able to recover all of the tokens held by `BToken` contracts. Because this action does not emit an event, the behavior will go unnoticed until there is damage such as financial loss.

Recommendations
Short term, add the missing event emission to `BetaBank.recover`.

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

6. Reliance on tx.origin to enforce EOA-only callers

Severity: Informational
Type: Access Controls
Target: `beta-contract/contracts/BetaRunnerBase.sol`

Difficulty: Medium
Finding ID: TOB-BFI-006

Description
Several contracts use the `onlyEOA` modifier from `BetaRunnerBase` to permit only externally owned accounts (EOAs) to initiate short selling and borrowing operations. To ensure that only EOA callers can initiate these operations, the modifier checks that `tx.origin == msg.sender`. However, the use of `tx.origin` is not futureproof; if EIP-3074 is implemented in a future network upgrade, the check it performs will be ineffective. Furthermore, this reliance on `tx.origin` negatively impacts users who rely on smart contract wallets.

The following contracts use the `onlyEOA` modifier, relying on `tx.origin`:

- `BetaRunnerBase`
- `BetaRunnerLending`
- `BetaRunnerUniswapV2`
- `BetaRunnerUniswapV3`

Exploit Scenario
The Beta Finance team deploys the `BetaRunner` contracts, with the assumption that only non-contract addresses will be able to interact with the system. EIP-3074 is implemented in a network upgrade. Despite the use of the `onlyEOA` modifier, smart contracts are now able to interact with the Beta Finance system , which may result in side effects for the security of the system.

Recommendations
Short term, consider the impacts of using the `onlyEOA` modifier, which may become ineffective in preventing contracts from interacting with the Beta Finance system.

Long term, design contracts in such a way that they function properly regardless of the type of address used to invoke the contracts.

7. Incorrectly sent tokens can be recovered by users

Severity: Medium | Difficulty: Low
Type: Data Validation | Finding ID: TOB-BFI-007
Target: `beta-contract/contracts/BetaRunnerUniswapV2/V3.sol`

Description
The `BetaRunnerUniswapV2` contract allows users to open and close short positions. When closing a position, a user can choose the amount of collateral he will recover. However, the code uses the `IERC20(col).balanceOf(address(this))` argument as the amount, which could allow users to recover incorrectly sent tokens.

```
/// @dev Continues the action (uniswap / sushiswap / pancakeswap)
 function _pairCallback(bytes calldata data) internal {
   CallbackData memory cb = abi.decode(data, (CallbackData));
   require(msg.sender == _pairFor(cb.path[0], cb.path[1]), '_pairCallback/bad-caller');
   uint len = cb.path.length;
   if (len > 2) {
      // [...]
   }
   if (cb.memo > 0) {
      // [...]
   } else {
     uint amountTake = uint(-cb.memo);
     (address und, address col) = (cb.path[len - 1], cb.path[0]);
     _repay(tx.origin, cb.pid, und, col, cb.amounts[len - 1], amountTake);
     IERC20(col).safeTransfer(msg.sender, cb.amounts[0]);
     _transferOut(col, tx.origin, IERC20(col).balanceOf(address(this)));
   }
 }
```

Figure 7.1: `BetaRunnerUniswapV2.sol#L104-L133`

To prevent this issue, use the Uniswap `swap` function, which calls the `BetaRunnerUniswapV2` contract's `uniswapV2Call` function, whose second and third arguments indicate how many tokens are sent and received. By using these arguments, the system can determine the correct amount of collateral to transfer.

Exploit Scenario
Alice incorrectly sends collateral tokens to the `BetaRunnerUniswapV2` contract. Bob closes his position (with the same collateral token as Alice's) and now receives both his tokens and Alice's tokens.

Recommendations
Short term, use the Uniswap `swap` function's callback arguments to calculate the exact collateral amount to send to users when they close short positions.

Long term, write properties and test them using Echidna.

8. Solidity compiler optimizations can be problematic

Severity: Informational
Type: Undefined Behavior
Target: `beta-contract`

Difficulty: Low
Finding ID: TOB-BFI-008

Description
The `beta-contract` project has enabled optional compiler optimizations in Solidity.

Because the codebase does not specify a Brownie configuration file, it uses the default configuration, which enables the Solidity optimizer with 200 runs.

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 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.

9. Risks associated with EIP-2612

Severity: Informational
Type: Configuration
Target: `beta-contract/contracts/BToken.sol`, `beta-contract/contracts/BetaToken.sol`

Difficulty: High
Finding ID: TOB-BFI-009

Description
The use of EIP-2612 increases the risk of `permit` function front-running as well as phishing attacks.

EIP-2612 uses signatures as an alternative to the traditional `approve` and `transferFrom` flow. These signatures allow a third party to transfer tokens on behalf of a user, with verification of a signed message.

An external party can front-run the `permit` function by submitting the signature first. The use of EIP-2612 makes it possible for a different party to front-run the initial caller's transaction. As a result, the intended caller's transaction will fail (as the signature has already been used and the funds have been transferred). This may also affect external contracts that rely on a successful `permit()` call for execution. For more information, see the EIP-2612 documentation.

EIP-2612 also makes it easier for an attacker to steal a user's tokens through phishing by asking for signatures in a context unrelated to the Beta Finance contracts. The hash message may look benign and random to the user.

Exploit Scenario
Bob has 1,000 `BETA` tokens. Eve creates a system meant to interact with the Beta Finance contracts by executing operations on a user's behalf. Her system should ask users to sign a hash to approve the Beta Finance contracts for the funds they want to interact with. She maliciously generates a hash to transfer 1,000 `BETA` tokens from Bob to herself instead of to the Beta Finance contracts. Eve asks Bob to sign the hash to approve the funds. Bob signs the hash, and Eve uses it to steal Bob's tokens.

Recommendations
Short term, document the risk of `permit` front-running and ensure that external contracts and scripts reflect this possibility. Alternatively, develop user documentation and on-chain mitigations to reduce the likelihood of a successful phishing campaign.

Long term, document best practices for users of the Beta Finance contracts. In addition to taking other precautions, users must do the following:

- Be extremely careful when signing a message
- Avoid signing messages from suspicious sources
- Always require hashing schemes to be public

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.

`libraries/SafeCast.sol`:

- Consider removing the unused `toUint160` and `toInt128` functions.

D. Token Integration Checklist

The following checklist provides recommendations for interacting 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 the checklist, you should have this 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` and `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.