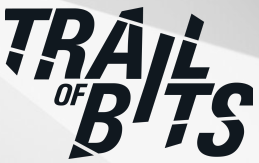

Franklin Templeton Tokenized Money Market Fund

Security Assessment

September 14, 2023

Prepared for:
Igor Natanzon
Franklin Templeton

Prepared by: **Tarun Bansal and Robert Schneider**

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 100+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

We maintain an exhaustive list of publications at https://github.com/trailofbits/publications, with links to papers, presentations, public audit reports, and podcast appearances.

In recent years, Trail of Bits consultants have showcased cutting-edge research through presentations at CanSecWest, HCSS, Devcon, Empire Hacking, GrrCon, LangSec, NorthSec, the O'Reilly Security Conference, PyCon, REcon, Security BSides, and SummerCon.

We specialize in software testing and code review projects, supporting client organizations in the technology, defense, and finance industries, as well as government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, and Zoom.

Trail of Bits also operates a center of excellence with regard to blockchain security. Notable projects include audits of Algorand, Bitcoin SV, Chainlink, Compound, Ethereum 2.0, MakerDAO, Matic, Uniswap, Web3, and Zcash.

To keep up to date with our latest news and announcements, please follow @trailofbits on Twitter and explore our public repositories at https://github.com/trailofbits. To engage us directly, visit our "Contact" page at https://www.trailofbits.com/contact, or email us at info@trailofbits.com.

Trail of Bits, Inc.
228 Park Ave S #80688
New York, NY 10003
https://www.trailofbits.com
info@trailofbits.com

Notices and Remarks

Copyright and Distribution

Test Coverage Disclaimer

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

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

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

Table of Contents

Executive Summary

Engagement Overview

Franklin Templeton engaged Trail of Bits to review the security of its Tokenized Money Market Fund (MMF) smart contracts. From May 1 to May 12, 2023, a team of two consultants conducted a security review of the client-provided source code, with four engineer-weeks of effort. Details of the project's timeline, test targets, and coverage are provided in subsequent sections of this report.

Project Scope

Our testing efforts were focused on the identification of flaws that could result in a compromise of confidentiality, integrity, or availability of the target system. We conducted this audit with full knowledge of the system. We had access to the source code and documentation. We performed static and dynamic automated and manual testing of the target system and its codebase, using both automated and manual processes.

Summary of Findings

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

EXPOSURE ANALYSIS

Severity	*Count*
High	4
Medium	3
Low	3
Informational	4
Undetermined	1

CATEGORY BREAKDOWN

Category	*Count*
Access Controls	1
Data Validation	9
Denial of Service	1
Patching	1
Timing	1
Undefined Behavior	2

Notable Findings

Significant flaws that impact system confidentiality, integrity, or availability are listed below.

- **TOB-FTMMF-01**
  Any shareholder can cancel any transaction request, which can result in a denial of service from the MMF system.

- **TOB-FTMMF-08**
  A single signature can be used multiple times to pass the threshold in the `MultiSigGenVerifier` contract, which could allow a single signer to take full control of the protocol.

- **TOB-FTMMF-11**
  Shareholders can stop the admin from deauthorizing them by front-running the `deauthorizeAccount` function in the `AuthorizationModule` contract.

Project Summary

Contact Information

The following managers were associated with this project:

Dan Guido, Account Manager
dan@trailofbits.com

Sam Greenup, Project Manager
sam.greenup@trailofbits.com

The following engineers were associated with this project:

Tarun Bansal, Consultant
tarun.bansal@trailofbits.com

Robert Schneider, Consultant
robert.schneider@trailofbits.com

Project Timeline

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

Date	Event
May 1, 2023	Pre-project kickoff call
May 8, 2023	Status update meeting #1
May 15, 2023	Delivery of report draft
May 15, 2023	Report readout meeting
June 23, 2023	Delivery of final report
September 14, 2023	Delivery of final report with fix review

Project Goals

The engagement was scoped to provide a security assessment of the Franklin Templeton Tokenized Money Market Fund (MMF) smart contracts. Specifically, we sought to answer the following non-exhaustive list of questions:

- Can an attacker make the system unusable?
- Can an attacker steal or freeze funds?
- Are there appropriate access controls in place for the different roles in the system?
- Is it possible to manipulate the system by front-running transactions?
- Does the system's behavior match the specification?
- Can users lose funds due to the system's mechanisms, such as buying and selling of shares, dividend distribution, and settlement of pending transactions?
- Can a non-shareholder user buy, sell, and keep shares?
- Can shareholders sell more shares than they bought?
- Can a signer take control of the protocol?
- Can an unauthorized user act on behalf of other users?
- Can administrators upgrade contracts when required?
- Does the distribution of dividends work as expected?

Project Targets

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

Money market fund

Filename `application---multichain---polygon-dev@e28accf8e6a.zip`

SHA-256 hash

`c3949796fe6ec623e6613b3797cc4cfd25d3e83152508825ebed4717b1cfe38f`

Type Solidity

Platform Polygon

Money market fund deployment scripts

Filename `dev-deployment.zip`

SHA-256 hash

`e971e46a1294dc422a8896951ebdd26f1ff4ba986ab5e5cbe6334f62d3dda862`

Type Typescript

Platform Polygon

Project Coverage

This section provides an overview of the analysis coverage of the review, as determined by our high-level engagement goals. Our approaches include the following:

- **General.** We reviewed the entire codebase for common Solidity flaws, such as missing contract existence checks on low-level calls, issues with access controls, unimplemented functions, and issues related to upgradeable contracts, such as state variable storage slot collision. We used Slither for static analysis of the code and Echidna for fuzz testing system invariants.

- **`MoneyMarketFund` token.** The `MoneyMarketFund` token is an ERC-20 token contract used to track users' shareholdings. It extends the ERC-20 token contract from the OpenZeppelin library. We manually reviewed it and its base contract, the `FundTokenBase` contract, for logic errors, mathematical errors, and access control issues. We also checked for ways to cause a denial of service (DoS) on the transaction settlement and dividends distribution processes.

- **`TransferAgentGateway` contract.** This contract serves as the main interaction point for users to interface with the system. The contract enforces access control rules on user actions and then forwards the calls to their respective modules. The `TransferAgentGateway` contract does not implement any business logic. We manually reviewed the `TransferAgentGateway` contract for correct configuration and access control issues.

- **`AuthorizationModule` contract.** The `AuthorizationModule` contract is used to manage access control roles for the different actors in the system. This contract extends the `AccessControlUpgradeable` contract from the OpenZeppelin library. We reviewed the code for correct initialization, logic issues, and access control issues. We checked public functions exposed by the base contracts to find ways to manipulate access roles in an unauthorized manner. We also looked for ways to front-run the admin actions to hinder correct administration of the fund.

- **`TransactionalModule` contract.** The system uses transaction requests to store users' buy/sell orders. These transaction requests are settled at the end of the day by the admins. The `TransactionalModule` contract is used to keep track of the transaction requests initiated by users and administrators. We manually reviewed it for logic errors (e.g., all pending transaction requests are processed on time), access control issues, and ways to reach an inconsistent system state.

- **`MultiSigGenVerifier` contract.** The `MultiSigGenVerifier` contract is a multi-party operated contract that is used to execute administrative actions. We

manually reviewed the signature verification code for issues related to use of invalid signatures, repeated signatures, and malleable signatures. We checked for protection from transaction replay attacks. We also looked for ways to add unauthorized signers and submitters to the contract.

- **Deployment scripts.** The deployment scripts are a set of seven TypeScript files, each containing code for the deployment of a specific module. We reviewed these scripts for the correct deployment and configuration of the contracts, along with the post-deployment steps to set up access control roles for all modules.

Coverage Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. The following list outlines the coverage limitations of the engagement and indicates system elements that may warrant further review:

- **Off-chain components:** The fund management system uses off-chain components to manage and automate administrative tasks. These components were out of scope, so they were not reviewed.
- **OpenZeppelin Defender:** The Franklin Templeton team uses OpenZeppelin Defender to deploy and manage their smart contracts. This is a third-party service and was out of scope for this engagement, so we did not review it.
- **Hardhat upgrades plugin:** The deployment scripts use a third-party plugin to deploy and initialize upgradable contracts in a secure manner. This plugin was out of scope, so we did not review it.

Automated Testing

Trail of Bits uses automated techniques to extensively test the security properties of software. We use both open-source static analysis and fuzzing utilities, along with tools developed in house, to perform automated testing of source code and compiled software.

Test Harness Configuration

We used the following tools in the automated testing phase of this project:

Tool	Description	Policy
Slither	A static analysis framework that can statically verify algebraic relationships between Solidity variables	Appendix D
Echidna	A smart contract fuzzer that can rapidly test security properties via malicious, coverage-guided test case generation	Appendix D

Areas of Focus

Our automated testing and verification work focused on the following system properties:

- Shareholders' holdings are equal to the amount they bought.
- Shareholders cannot sell more shares than they bought.
- An unauthorized user cannot buy, sell, or keep shares.
- Dividend distribution changes holdings as expected.

Test Results

The results of this focused testing are detailed below.

Global system invariants: We tested the following expected system properties.

Property	Tool	Result
Users cannot sell more shares than they bought.	Echidna	**Passed**

The `MoneyMarketFund` contract's holdings are equal to the amount purchased after a buy transaction is settled. Note: The rounding direction is in favor of the protocol, so no issue was created for this failure.	Echidna	**FAILED**
An account's balance does the following: • Increases when a positive rate dividend is paid • Decreases when a negative rate dividend is paid • Does not change when a zero rate dividend is paid	Echidna	**FAILED** TOB-FTMMF-15
The number of accounts with shares does not exceed the total number of authorized accounts.	Echidna	**FAILED** TOB-FTMMF-04

Codebase Maturity Evaluation

Trail of Bits uses a traffic-light protocol to provide each client with a clear understanding of the areas in which its codebase is mature, immature, or underdeveloped. Deficiencies identified here often stem from root causes within the software development life cycle that should be addressed through standardization measures (e.g., the use of common libraries, functions, or frameworks) or training and awareness programs.

Category	Summary	Result
Arithmetic	The codebase does not rely on complex arithmetic, limiting most of the potential risks. However, the system lacks a systematic approach toward explicit rounding behavior (see appendix G for recommendations). An automated testing tool (e.g., fuzzing) would also help evaluate the loss of precision risks.	**Moderate**
Auditing	Critical state-changing functions, such as updating signers on the administrative multisignature wallet, do not emit events. The existing off-chain monitoring system is focused on the correctness of user balances. Additionally, no incident response plan has been prepared, which would make the recovery process challenging in the event of a compromise (see appendix H).	**Weak**
Authentication / Access Controls	Access controls in the system are dispersed across multiple contracts, which makes chained functions difficult to follow. Each module stores its owner independently, and changing one module's owner will make multiple privileged actors control different parts of the system. This makes upgrading the system difficult. Similarly, having access control checks divided into multiple places may lead to issues such as TOB-FTMMF-01. The following steps can help ensure strict access controls: 1. Follow the principle of least privilege.	**Moderate**

	2. Minimize privileged actors controlling the same function on different components. 3. Document all system actors and their intended access levels. 4. Implement a test suite with a focus on negative cases and edge cases to test failure of unauthorized actions.	
Complexity Management	In general, code is structured well with small functions that are easy to test. However, the codebase has issues with code duplication and unused code, as detailed below: 1. Duplicate code is used in functions (e.g., the same access control checks are used in all functions of the `FundBaseToken` contract). 2. The body of a function is copied into another function (e.g., the `_updateHoldingsSet` function is copied into the `updateHolderInList` function). 3. The same constants (e.g., module IDs and roles) are defined in multiple contracts. 4. Unused code exists in the codebase (e.g., the `FundBaseToken` contract extends the `AccessControlEnumerable` contract). Documenting system components, their behavior, and interactions among them can help reduce duplicate code.	**Weak**
Decentralization	The system is not meant to be decentralized and has several points of centralization (e.g., the privileged actors have full control over the users' actions and balances).	**Not Applicable**
Documentation	The documentation is a high-level specification of how the system works and includes diagrams for use cases. Some parts of the documentation are not complete or up to date with the implementation. The source code uses NatSpec documentation comments, but not all functions are commented on, and some functions' inline comments lack detail.	**Moderate**

	The lack of documentation consistency and implementation details impedes understanding the system and increases the likelihood of mistakes being introduced in later updates.	
Front-Running Resistance	User actions on the MMF are resistant to general maximum extractable value (MEV) risks because all users' buy and sell requests are settled by the admin at a fixed price, and transfers are restricted. The MMF token is fully controlled, with only the admin having the authority to mint and burn it, using a price determined off-chain. However, further work could be done to ensure admin operations are protected against front-running risks. Finally, the documentation and tests lack a focus on front-running risks.	**Moderate**
Low-Level Manipulation	The use of assembly and low-level calls is limited and justified. However, additional inline documentation, with a focus on low-level and assembly risks, would be beneficial. Finally, no tests specifically targeted EVM edge cases, increasing the likelihood of mistakes.	**Moderate**
Testing and Verification	The current test suite covers both positive and negative cases; however, the coverage is insufficient. We recommend that the Franklin Templeton team spend additional development time updating their tests to achieve higher coverage of edge cases. Additionally, the codebase would benefit from the inclusion of property testing, which would help verify critical system invariants. Once the properties and invariants have been documented, they can be used as a baseline to improve the unit tests and, in the future, to include automated testing techniques such as fuzzing.	**Moderate**

Summary of Findings

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

ID	Title	Type	Severity
1	Canceling all transaction requests causes DoS on MMF system	Access Controls	High
2	Lack of validation in the IntentValidationModule contract can lead to inconsistent state	Data Validation	High
3	Pending transactions cannot be settled	Data Validation	Medium
4	Deauthorized accounts can keep shares of the MMF	Data Validation	Medium
5	Solidity compiler optimizations can be problematic	Undefined Behavior	Informational
6	Project dependencies contain vulnerabilities	Patching	Undetermined
7	Unimplemented getVersion function returns default value of zero	Undefined Behavior	Informational
8	The MultiSigGenVerifier threshold can be passed with a single signature	Data Validation	High
9	Shareholders can renounce their authorization role	Data Validation	Low
10	Risk of multiple dividend payouts in a day	Data Validation	Medium
11	Shareholders can stop admin from deauthorizing them	Timing	High

12	Total number of submitters in MultiSigGenVerifier contract can be more than allowed limit of MAX_SUBMITTERS	Data Validation	**Informational**
13	Lack of contract existence check on target address	Data Validation	**Low**
14	Pending transactions can trigger a DoS	Denial of Service	**Informational**
15	Dividend distribution has an incorrect rounding direction for negative rates	Data Validation	**Low**

Detailed Findings

1. Canceling all transaction requests causes DoS on MMF system

Severity: **High**	Difficulty: **Low**
Type: Access Controls	Finding ID: TOB-FTMMF-01
Target: `FT/TransferAgentGateway.sol`, `FT/infrastructure/modules/TransactionalModule.sol`	

Description

Any shareholder can cancel any transaction request, which can result in a denial of service (DoS) from the MMF system.

The `TransactionalModule` contract uses transaction requests to store buy and sell orders from users. These requests are settled at the end of the day by the admins. Admins can create or cancel a request for any user. Users can create requests for themselves and cancel their own requests.

The `TransferAgentGateway` contract is an entry point for all user and admin actions. It implements access control checks and forwards the calls to their respective modules. The `cancelRequest` function in the `TransferAgentGateway` contract checks that the caller is the owner or a shareholder. However, if the caller is not the owner, the caller is not matched against the `account` argument. This allows any shareholder to call the `cancelRequest` function in the `TransactionalModule` for any `account` and `requestId`.

```
function cancelRequest(
    address account,
    bytes32 requestId,
    string calldata memo
) external override {
    require(
        msg.sender == owner() ||
            IAuthorization(
                moduleRegistry.getModuleAddress(AUTHORIZATION_MODULE)
            ).isAccountAuthorized(msg.sender),
        "OPERATION_NOT_ALLOWED_FOR_CALLER"
    );
```

```
    ICancellableTransaction(
        moduleRegistry.getModuleAddress(TRANSACTIONAL_MODULE)
    ).cancelRequest(account, requestId, memo);
}
```

Figure 1.1: The `cancelRequest` function in the `TransferAgentGateway` contract

As shown in figure 1.2, the `if` condition in the `cancelRequest` function in the `TransactionalModule` contract implements a check that does not allow shareholders to cancel transaction requests created by the admin. However, this check passes because the `TransferAgentGateway` contract is set up as the admin account in the authorization module.

```
function cancelRequest(
    address account,
    bytes32 requestId,
    string calldata memo
) external override onlyAdmin onlyShareholder(account) {
    require(
        transactionDetailMap[requestId].txType >
            ITransactionStorage.TransactionType.INVALID,
        "INVALID_TRANSACTION_ID"
    );

    if (!transactionDetailMap[requestId].selfService) {
        require(
            IAuthorization(modules.getModuleAddress(AUTHORIZATION_MODULE))
                .isAdminAccount(msg.sender),
            "CALLER_IS_NOT_AN_ADMIN"
        );
    }

    require(
        pendingTransactionsMap[account].remove(requestId),
        "INVALID_TRANSACTION_ID"
    );
    delete transactionDetailMap[requestId];
    accountsWithTransactions.remove(account);

    emit TransactionCancelled(account, requestId, memo);
}
```

Figure 1.2: The `cancelRequest` function in the `TransactionalModule` contract

Thus, a shareholder can cancel any transaction request created by anyone.

Exploit Scenario

Eve becomes an authorized shareholder and sets up a bot to listen to the `TransactionSubmitted` event on the `TransactionalModule` contract. The bot calls the `cancelRequest` function on the `TransferAgentGateway` contract for every event and

cancels all the transaction requests before they are settled, thus executing a DoS attack on the MMF system.

Recommendations

Short term, add a check in the `TransferAgentGateway` contract to allow shareholders to cancel requests only for their own accounts.

Long term, document access control rules in a publicly accessible location. These rules should encompass admin, non-admin, and common functions. Ensure the code adheres to that specification by extending unit test coverage for positive and negative expectations within the system. Add fuzz tests where access control rules are the invariants under test.

2. Lack of validation in the IntentValidationModule contract can lead to inconsistent state

Severity: **High**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-FTMMF-02
Target: `FT/infrastructure/modules/IntentValidationModule.sol`	

Description

Lack of validation in the state-modifying functions of the `IntentValidationModule` contract can cause users to be locked out of the system.

As shown in figure 2.1, the `setDeviceKey` function in `IntentValidationModule` allows adding a device ID and key to multiple accounts, which may result in the unauthorized use of a device ID.

```
function setDeviceKey(
    address account,
    uint256 deviceId,
    string memory key
) external override onlyAdmin {
    devicesMap[account].add(deviceId);
    deviceKeyMap[deviceId] = key;

    emit DeviceKeyAdded(account, deviceId);
}
```

Figure 2.1: The `setDeviceKey` functions in the `IntentValidationModule` contract

Additionally, a lack of validation in the `clearDeviceKey` and `clearAccountKeys` functions can cause the key for a device ID to become zero, which may prevent users from authenticating their requests.

```
function clearDeviceKey(
    address account,
    uint256 deviceId
) external override onlyAdmin {
    _removeDeviceKey(account, deviceId);
}

function clearAccountKeys(address account) external override onlyAdmin {
    uint256[] memory devices = devicesMap[account].values();
    for (uint i = 0; i < devices.length; ) {
        _removeDeviceKey(account, devices[i]);
```

```
        unchecked {
            i++;
        }
    }
}
```

Figure 2.2: Functions to clear device ID and key in the `IntentValidationModule` contract

The account-to–device ID mapping and device ID–to-key mapping are used to authenticate user actions in an off-chain component, which can malfunction in the presence of these inconsistent states and lead to the authentication of malicious user actions.

Exploit Scenario

An admin adds the `DEV_A` device and the `KEY_K` key to Bob. Then there are multiple scenarios to cause an inconsistent state, such as the following:

Adding one device to multiple accounts:

1. An admin adds the `DEV_A` device and the `KEY_K` key to Alice by mistake.
2. Alice can use Bob's device to send unauthorized requests.

Overwriting a key for a device ID:

1. An admin adds the `DEV_A` device and the `KEY_L` key to Alice, which overwrites the key for the `DEV_A` device from `KEY_K` to `KEY_L`.
2. Bob cannot authenticate his requests with his `KEY_K` key.

Setting a key to zero for a device ID:

1. An admin adds the `DEV_A` device and the `KEY_K` key to Alice by mistake.
2. An admin removes the `DEV_A` device from Alice's account. This sets the key for the `DEV_A` device to zero, which is still added to Bob's account.
3. Bob cannot authenticate his requests with his `KEY_K` key.

Recommendations

Short term, make the following changes:

- Add a check in the `setDeviceKey` function to ensure that a device is not added to multiple accounts.
- Add a new function to update the key of an already added device with correct validation checks for the update.

Long term, document valid system states and the state transitions allowed from each state. Ensure proper data validation checks are added in all state-modifying functions with unit and fuzzing tests.

3. Pending transactions cannot be settled

Severity: **Medium**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-FTMMF-03
Target: `FT/infrastructure/modules/TransactionalModule.sol`, `FT/infrastructure/modules/TransferAgentModule.sol`, `FT/MoneyMarketFund.sol`	

Description

An account removed from the `accountsWithTransactions` state variable will have its pending transactions stuck in the system, resulting in an opportunity cost loss for the users.

The `accountsWithTransactions` state variable in the `TransactionalModule` contract is used to keep track of accounts with pending transactions. It is used in the following functions:

- The `getAccountsWithTransactions` function to return the list of accounts with pending transactions
- The `hasTransactions` function to check if an account has pending transactions.

However, the `cancelRequest` function in the `TransactionalModule` contract removes the account from the `accountsWithTransactions` list for every cancellation. If an account has multiple pending transactions, canceling only one of the transaction requests will remove the account from the `accountsWithTransactions` list.

```
function cancelRequest(
    address account,
    bytes32 requestId,
    string calldata memo
) external override onlyAdmin onlyShareholder(account) {
    require(
        transactionDetailMap[requestId].txType >
            ITransactionStorage.TransactionType.INVALID,
        "INVALID_TRANSACTION_ID"
    );

    if (!transactionDetailMap[requestId].selfService) {
        require(
            IAuthorization(modules.getModuleAddress(AUTHORIZATION_MODULE))
```

```
                .isAdminAccount(msg.sender),
            "CALLER_IS_NOT_AN_ADMIN"
        );
    }

    require(
        pendingTransactionsMap[account].remove(requestId),
        "INVALID_TRANSACTION_ID"
    );
    delete transactionDetailMap[requestId];
    accountsWithTransactions.remove(account);

    emit TransactionCancelled(account, requestId, memo);
}
```

Figure 3.1: The `cancelRequest` function in the `TransactionalModule` contract

In figure 3.1, the account has pending transactions, but it is not present in the `accountsWithTransactions` list. The off-chain components and other functionality relying on the `getAccountsWithTransactions` and `hasTransactions` functions will see these accounts as not having any pending transactions. This may result in non-settlement of the pending transactions for these accounts, leading to a loss for the users.

Exploit Scenario

Alice, a shareholder, creates multiple transaction requests and cancels the last request. For the next settlement process, the off-chain component calls the `getAccountsWithTransactions` function to get the list of accounts with pending transactions and settles these accounts. After the settlement run, Alice checks her balance and is surprised that her transaction requests are not settled. She loses profits from upcoming market movements.

Recommendations

Short term, have the code use the `unlistFromAccountsWithPendingTransactions` function in the `cancelRequest` function to update the `accountsWithTransactions` list.

Long term, document the system state machine specification and follow it to ensure proper data validation checks are added in all state-modifying functions.

4. Deauthorized accounts can keep shares of the MMF

Severity: **Medium**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-FTMMF-04
Target: `FT/infrastructure/modules/AuthorizationModule.sol`	

Description

An unauthorized account can keep shares if the admin deauthorizes the shareholder without zeroing their balance. This can lead to legal issues because unauthorized users can keep shares of the MMF.

The `deauthorizeAccount` function in the `AuthorizationModule` contract does not check that the balance of the provided account is zero before revoking the `ROLE_FUND_AUTHORIZED` role:

```
function deauthorizeAccount(
    address account
) external override onlyRole(ROLE_AUTHORIZATION_ADMIN) {
    require(account != address(0), "INVALID_ADDRESS");
    address txModule = modules.getModuleAddress(
        keccak256("MODULE_TRANSACTIONAL")
    );
    require(txModule != address(0), "MODULE_REQUIRED_NOT_FOUND");
    require(
        hasRole(ROLE_FUND_AUTHORIZED, account),
        "SHAREHOLDER_DOES_NOT_EXISTS"
    );
    require(
        !ITransactionStorage(txModule).hasTransactions(account),
        "PENDING_TRANSACTIONS_EXIST"
    );

    _revokeRole(ROLE_FUND_AUTHORIZED, account);
    emit AccountDeauthorized(account);
}
```

Figure 4.1: The `deauthorizeAccount` function in the `AuthorizationModule` contract

If an admin account deauthorizes a shareholder account without making the balance zero, the unauthorized account will keep the shares of the MMF. The impact is limited, however, because the unauthorized account will not be able to liquidate the shares. The admin can also adjust the balance of the account to make it zero. However, if the admin forgets to

adjust the balance or is unable to adjust the balance, it can lead to an unauthorized account holding shares of the MMF.

Recommendations

Short term, add a check in the `deauthorizeAccount` function to ensure that the balance of the provided account is zero.

Long term, document the system state machine specification and follow it to ensure proper data validation checks are added in all state-modifying functions. Add fuzz tests where the rules enforced by those validation checks are the invariants under test.

5. Solidity compiler optimizations can be problematic

Severity: **Informational**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-FTMMF-05
Target: `./hardhat.config.js`	

Description
The MMF has enabled optional compiler optimizations in Solidity. According to a November 2018 audit of the Solidity compiler, the optional optimizations may not be safe.

```
optimizer: {
  enabled: true,
  runs: 200,
},
```

Figure 5.1: Hardhat optimizer enabled in `hardhat.config.js`

Solidity compiler optimizations are disabled by default, and it is unclear how many contracts in the wild use them. Therefore, it is unclear how well they are being tested and exercised. Moreover, optimizations are actively being developed.

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 Keccak-256 was reported.

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 MMF 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.

6. Project dependencies contain vulnerabilities

Severity: **Undetermined**	Difficulty: **High**
Type: Patching	Finding ID: TOB-FTMMF-06
Target: `./package.json`	

Description

Although dependency scans did not identify a direct threat to the project codebase, `npm audit` found 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 MMF system. The output detailing the identified issues has been included below:

Dependency	Version	ID	Description
`flat`	<5.0.1	CVE-2020-36632	`flat` vulnerable to prototype pollution
`@openzeppelin/contracts`	3.2.0 - 4.8.2	CVE-2023-30541	OpenZeppelin contracts' `TransparentUpgradeableProxy` clashing selector calls may not be delegated
`@openzeppelin/contracts-upgradeable`	>= 3.2.0, < 4.8.3	CVE-2023-30541	OpenZeppelin contracts' `TransparentUpgradeableProxy` clashing selector calls may not be delegated
`minimatch`	< 3.0.5	CVE-2022-3517	`minimatch` ReDoS vulnerability
`request`	<= 2.88.2	CVE-2023-28155	Server-side request forgery in `request`

Table 6.1: npm audit output

Exploit Scenario

Alice installs the dependencies for this project on a clean machine. Unbeknownst to Alice, a dependency of the project has become malicious or exploitable. Alice subsequently uses the vulnerable 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 dependencies cannot be updated when a vulnerability is disclosed, ensure that the project codebase does not use and is not affected by the dependency's vulnerable functionality.

7. Unimplemented getVersion function returns default value of zero

Severity: **Informational**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-FTMMF-07
Target: `FT/infrastructure/modules/TransferAgentModule.sol`	

Description
The `getVersion` function within the `TransferAgentModule` contract is not implemented; at present, it yields the default `uint8` value of zero.

```
function getVersion() external pure virtual override returns (uint8) {}
```

Figure 7.1: Unimplemented `getVersion` function in the `TransferAgentModule` contract

The other module contracts establish a pattern where the `getVersion` function is supposed to return a value of one.

```
function getVersion() external pure virtual override returns (uint8) {
  return 1;
}
```

Figure 7.2: Implemented `getVersion` function in the `TransactionalModule` contract

Exploit Scenario
Alice calls the `getVersion` function on the `TransferAgentModule` contract. It returns zero, and all the other module contracts return one. Alice misunderstands the system and which contracts are on what version of their lifecycle.

Recommendations
Short term, implement the `getVersion` function in the `TransferAgentModule` contract so it matches the specification established in the other modules.

Long term, use the Slither static analyzer to catch common issues such as this one. Implement `slither-action` into the project's CI pipeline.

8. The MultiSigGenVerifier threshold can be passed with a single signature

Severity: **High**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-FTMMF-08
Target: `FT/infrastructure/multisig/MultiSigGenVerifier.sol`	

Description

A single signature can be used multiple times to pass the threshold in the `MultiSigGenVerifier` contract, allowing a single signer to take full control of the system.

The `signedDataExecution` function in the `MultiSigGenVerifier` contract verifies provided signatures and accumulates the `acquiredThreshold` value in a loop as shown in figure 8.1:

```
for (uint256 i = 0; i < signaturesCount; i++) {
    (v, r, s) = _splitSignature(signatures, i);
    address signerRecovered = ecrecover(hash, v, r, s);

    if (signersSet.contains(signerRecovered)) {
        acquiredThreshold += signersMap[signerRecovered];
    }
}
```

Figure 8.1: The signer recovery section of the `signedDataExecution` function in the `MultiSigGenVerifier` contract

This code checks whether the recovered signer address is one of the previously added signers and adds the signer's weight to `acquiredThreshold`. However, the code does not check that all the recorded signers are unique, which allows the submitter to pass the threshold with only a single signature to execute the signed transaction.

The current function has an implicit zero-address check in the subsequent `if` statement—to add new signers, they must not be `address(0)`. If this logic changes in the future, the impact of the `ecrecover` function returning `address(0)` (which happens when a signature is malformed) must be carefully reviewed.

Exploit Scenario

Eve, a signer, colludes with a submitter to settle their transactions at a favorable date and price. Eve signs the transaction and provides it to the submitter. The submitter uses this signature to call the `signedDataExecution` function by repeating the same signature

multiple times in the `signatures` argument array to pass the threshold. Using this method, Eve can execute any admin transaction without consent from other admins.

Recommendations

Short term, have the code verify that the signatures provided to the `signedDataExecution` function are unique. One way of doing this is to sort the signatures in increasing order of the signer addresses and verify this order in the loop. An example of this order verification code is shown in figure 8.2:

```
address lastSigner = address(0);
for (uint256 i = 0; i < signaturesCount; i++) {
    (v, r, s) = _splitSignature(signatures, i);
    address signerRecovered = ecrecover(hash, v, r, s);

    require(lastSigner < signerRecovered);
    lastSigner = signerRecovered;

    if (signersSet.contains(signerRecovered)) {
        acquiredThreshold += signersMap[signerRecovered];
    }
}
```

Figure 8.2: An example code to verify uniqueness of the provided signatures

Long term, expand unit test coverage to account for common edge cases, and carefully consider all possible values for any user-provided inputs. Implement fuzz testing to explore complex scenarios and find difficult-to-detect bugs in functions with user-provided inputs.

9. Shareholders can renounce their authorization role	
Severity: **Low**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-FTMMF-09
Target: `FT/infrastructure/modules/AuthorizationModule.sol`	

Description

Shareholders can renounce their authorization role. As a result, system contracts that check for authorization and off-chain components may not work as expected because of an inconsistent system state.

The `AuthorizationModule` contract extends the `AccessControlUpgradeable` contract from the OpenZeppelin library. The `AccessControlUpgradeable` contract has a public `renounceRole` function, which can be called by anyone to revoke a role on their own account.

```
function renounceRole(bytes32 role, address account) public virtual override {
    require(account == _msgSender(), "AccessControl: can only renounce roles for
self");

    _revokeRole(role, account);
}
```

Figure 9.1: The `renounceRole` function of the base contract from the OpenZeppelin library

Any shareholder can use the `renounceRole` function to revoke the `ROLE_FUND_AUTHORIZED` role on their own account without authorization from the admin. This role is used in three functions in the `AccessControlUpgradeable` contract:

1. The `isAccountAuthorized` function to check if an account is authorized
2. The `getAuthorizedAccountsCount` to get the number of authorized accounts
3. The `getAuthorizedAccountAt` to get the authorized account at an index

Other contracts and off-chain components relying on these functions may find the system in an inconsistent state and may not be able to work as expected.

Exploit Scenario

Eve, an authorized shareholder, renounces her `ROLE_FUND_AUTHORIZED` role. The off-chain components fetch the number of authorized accounts, which is one less than the expected value. The off-chain component is now operating on an inaccurate contract state.

Recommendations

Short term, have the code override the `renounceRole` function in the `AuthorizationModule` contract. Make this overridden function an admin-only function.

Long term, read all the library code to find public functions exposed by the base contracts and override them to implement correct business logic and enforce proper access controls. Document any changes between the original OpenZeppelin implementation and the MMF implementation. Be sure to thoroughly test overridden functions and changes in unit tests and fuzz tests.

10. Risk of multiple dividend payouts in a day

Severity: **Medium**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-FTMMF-10
Target: `FT/infrastructure/modules/TransferAgentModule.sol`, `FT/MoneyMarketFund.sol`	

Description

The fund manager can lose the system's money by making multiple dividend payouts in a day when they should be paid out only once a day.

The `distributeDividends` function in the `MoneyMarketFund` contract takes the `date` as an argument. This `date` value is not validated to be later than the date from an earlier execution of the `distributeDividends` function.

```
function distributeDividends(
    address[] calldata accounts,
    uint256 date,
    int256 rate,
    uint256 price
)
    external
    onlyAdmin
    onlyWithValidRate(rate)
    onlyValidPaginationSize(accounts.length, MAX_ACCOUNT_PAGE_SIZE)
{
    lastKnownPrice = price;
    for (uint i = 0; i < accounts.length; ) {
        _processDividends(accounts[i], date, rate, price);
        unchecked {
            i++;
        }
    }
}
```

Figure 10.1: The `distributeDividends` function in the `MoneyMarketFund` contract

As a result, the admin can distribute dividends multiple times a day, which will result in the loss of funds from the company to the users. The admin can correct this mistake by using the `adjustBalance` function, but adjusting the balance for all the system users will be a difficult and costly process.

The same issue also affects the following three functions:

1. The `endOfDay` function in the `MoneyMarketFund` contract
2. The `distributeDividends` function in the `TransferAgentModule` contract
3. The `endOfDay` function in the `TransferAgentModule` contract.

Exploit Scenario

The admin sends a transaction to distribute dividends. The transaction is not included in the blockchain because of congestion or gas estimation errors. Forgetting about the earlier transaction, the admin sends another transaction, and both transactions are executed to distribute dividends on the same day.

Recommendations

Short term, have the code store the last dividend distribution date and validate that the `date` argument in all the dividend distribution functions is later than the last stored dividend date.

Long term, document the system state machine specification and follow it to ensure proper data validation checks are added to all state-modifying functions.

11. Shareholders can stop admin from deauthorizing them

Severity: **High**	Difficulty: **Medium**
Type: Timing	Finding ID: TOB-FTMMF-11
Target: `FT/infrastructure/modules/AuthorizationModule.sol`	

Description

Shareholders can prevent the admin from deauthorizing them by front-running the `deauthorizeAccount` function in the `AuthorizationModule` contract.

The `deauthorizeAccount` function reverts if the provided account has one or more pending transactions.

```
function deauthorizeAccount(
    address account
) external override onlyRole(ROLE_AUTHORIZATION_ADMIN) {
    require(account != address(0), "INVALID_ADDRESS");
    address txModule = modules.getModuleAddress(
        keccak256("MODULE_TRANSACTIONAL")
    );
    require(txModule != address(0), "MODULE_REQUIRED_NOT_FOUND");
    require(
        hasRole(ROLE_FUND_AUTHORIZED, account),
        "SHAREHOLDER_DOES_NOT_EXISTS"
    );
    require(
        !ITransactionStorage(txModule).hasTransactions(account),
        "PENDING_TRANSACTIONS_EXIST"
    );

    _revokeRole(ROLE_FUND_AUTHORIZED, account);
    emit AccountDeauthorized(account);
}
```

Figure 11.1: The `deauthorizeAccount` function in the `AuthorizationModule` contract

A shareholder can front-run a transaction executing the `deauthorizeAccount` function for their account by submitting a new transaction request to buy or sell shares. The `deauthorizeAccount` transaction will revert because of a pending transaction for the shareholder.

Exploit Scenario
Eve, a shareholder, sets up a bot to front-run all `deauthorizeAccount` transactions that add a new transaction request for her. As a result, all admin transactions to deauthorize Eve fail.

Recommendations
Short term, remove the check for the pending transactions of the provided account and consider one of the following:

1. Have the code cancel the pending transactions of the provided account in the `deauthorizeAccount` function.
2. Add a check in the `_processSettlements` function in the `MoneyMarketFund` contract to skip unauthorized accounts. Add the same check in the `_processSettlements` function in the `TransferAgentModule` contract.

Long term, always analyze all contract functions that can be affected by attackers front-running calls to manipulate the system.

12. Total number of submitters in MultiSigGenVerifier contract can be more than allowed limit of MAX_SUBMITTERS

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-FTMMF-12
Target: `FT/infrastructure/multisig/MultiSigGenVerifier.sol`	

Description

The total number of submitters in the `MultiSigGenVerifier` contract can be more than the allowed limit of `MAX_SUBMITTERS`.

The `addSubmitters` function in the `MultiSigGenVerifier` contract does not check that the total number of submitters in the `submittersSet` is less than the value of the `MAX_SUBMITTERS` constant.

```
function addSubmitters(address[] calldata submitters) public onlyVerifier {
    require(submitters.length <= MAX_SUBMITTERS, "INVALID_ARRAY_LENGTH");
    for (uint256 i = 0; i < submitters.length; i++) {
        submittersSet.add(submitters[i]);
    }
}
```

Figure 12.1: The `addSubmitters` function in the `MultiSigGenVerifier` contract

This allows the admin to add more than the maximum number of allowed submitters to the `MultiSigGenVerifier` contract.

Recommendations

Short term, add a check to the `addSubmitters` function to verify that the length of the `submittersSet` is less than or equal to the `MAX_SUBMITTERS` constant.

Long term, document the system state machine specification and follow it to ensure proper data validation checks are added in all state-modifying functions. To ensure `MAX_SUBMITTERS` is never exceeded, add fuzz testing where `MAX_SUBMITTERS` is the system invariant under test.

13. Lack of contract existence check on target address

Severity: **Low**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-FTMMF-13
Target: `FT/infrastructure/multisig/MultiSigGenVerifier.sol`	

Description

The `signedDataExecution` function lacks validation to ensure that the `target` argument is a contract address and not an externally owned account (EOA). The absence of such a check could lead to potential security issues, particularly when executing low-level calls to an address not containing contract code.

Low-level calls to an EOA return `true` for the `success` variable instead of reverting as they would with a contract address. This unexpected behavior could trigger inadvertent execution of subsequent code relying on the `success` variable to be accurate, potentially resulting in undesired outcomes. The `onlySubmitter` modifier limits the potential impact of this vulnerability.

```
function signedDataExecution(
        address target,
        bytes calldata payload,
        bytes calldata signatures
    ) external onlySubmitter {

        ...

        // Wallet logic
        if (acquiredThreshold >= _getRequiredThreshold(target)) {
            (bool success, bytes memory result) = target.call{value: 0}(
                payload
            );

            emit TransactionExecuted(target, result);

            if (!success) {
                assembly {
                    result := add(result, 0x04)
                }
                revert(abi.decode(result, (string)));
            }
        } else {
            revert("INSUFICIENT_THRESHOLD_ACQUIRED");
        }
```

```
    }
```

Figure 13.1: The `signedDataExecution` function in the `MultiSigGenVerifier` contract

Exploit Scenario

Alice, an authorized submitter account, calls the `signedDataExecution` function, passing in an EOA address instead of the expected contract address. The low-level call to the `target` address returns successfully and does not revert. As a result, Alice thinks she has executed code but in fact has not.

Recommendations

Short term, integrate a contract existence check to ensure that code is present at the address passed in as the `target` argument.

Long term, use the Slither static analyzer to catch issues such as this one. Consider integrating `slither-action` into the project's CI pipeline.

14. Pending transactions can trigger a DoS

Severity: **Informational**	Difficulty: **Medium**
Type: Denial of Service	Finding ID: TOB-FTMMF-14
Target: `FT/infrastructure/modules/TransferAgentModule.sol`, `FT/MoneyMarketFund.sol`	

Description

An unbounded number of pending transactions can cause the `_processSettlements` function to run out of gas while trying to process them.

There is no restriction on the length of pending transactions a user might have, and gas-intensive operations are performed in the for-loop of the `_processSettlements` function. If an account returns too many pending transactions, operations that call `_processSettlements` might revert with an out-of-gas error.

```
function _processSettlements(
    address account,
    uint256 date,
    uint256 price
) internal whenTransactionsExist(account) {
    bytes32[] memory pendingTxs = ITransactionStorage(
        moduleRegistry.getModuleAddress(TRANSACTIONAL_MODULE)
    ).getAccountTransactions(account);
    for (uint256 i = 0; i < pendingTxs.length; ) {
        ...
```

Figure 14.1: The `pendingTxs` loop in the `_processSettlements` function in the `MoneyMarketFund` contract

The same issue affects the `_processSettlements` function in the `TransferAgentModule` contract.

Exploit Scenario

Eve submits multiple transactions to the `requestSelfServiceCashPurchase` function, and each creates a pending transaction record in the `pendingTransactionsMap` for Eve's account. When `settleTransactions` is called with an array of accounts that includes Eve, the `_processSettlements` function tries to process all her pending transactions and runs out of gas in the attempt.

Recommendations

Short term, make the following changes to the transaction settlement flow:

1. Enhance the off-chain component of the system to identify accounts with too many pending transactions and exclude them from calls to `_processSettlements` flows.
2. Create another transaction settlement function that paginates over the list of pending transactions of a single account.

Long term, implement thorough testing protocols for these loop structures, simulating various scenarios and edge cases that could potentially result in unbounded inputs. Ensure that all loop structures are robustly designed with safeguards in place, such as constraints and checks on input variables.

15. Dividend distribution has an incorrect rounding direction for negative rates

Severity: **Low**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-FTMMF-15
Target: `FT/infrastructure/modules/TransferAgentModule.sol`, `FT/MoneyMarketFund.sol`	

Description

The rounding direction of the dividend calculation in the `_processDividends` function benefits the user when the dividend rate is negative, causing the fund to lose value it should retain.

The division operation that computes dividend shares is rounding down in the `_processDividends` function of the `MoneyMarketFund` contract:

```
function _processDividends(
    address account,
    uint256 date,
    int256 rate,
    uint256 price
) internal whenHasHoldings(account) {
    uint256 dividendAmount = balanceOf(account) * uint256(abs(rate));
    uint256 dividendShares = dividendAmount / price;

    _payDividend(account, rate, dividendShares);
    // handle very unlikely scenario if occurs
    _handleNegativeYield(account, rate, dividendShares);
    _removeEmptyAccountFromHoldingsSet(account);

    emit DividendDistributed(account, date, rate, price, dividendShares);
}
```

Figure 15.1: The `_processDividends` function in the `MoneyMarketFund` contract

As a result, for a negative dividend rate, the rounding benefits the user by subtracting a lower number of shares from the user balance. In particular, if the rate is low and the price is high, the dividend can round down to zero.

The same issue affects the `_processDividends` function in the `TransferAgentModule` contract.

Exploit Scenario

Eve buys a small number of shares from multiple accounts. The dividend rounds down and is equal to zero. As a result, Eve avoids the losses from the downside movement of the fund while enjoying profits from the upside.

Recommendations

Short term, have the `_processDividends` function round up the number of `dividendShares` for negative dividend rates.

Long term, document the expected rounding direction for every arithmetic operation (see appendix G) and follow it to ensure that rounding is always beneficial to the fund. Use Echidna to find issues arising from the wrong rounding direction.

Summary of Recommendations

The Franklin Templeton Tokenized Money Market Fund is a work in progress with multiple planned iterations. Trail of Bits recommends that Franklin Templeton address the findings detailed in this report and take the following additional steps prior to deployment:

- Implement fix recommendations in the `TransferAgentGateway` contract, `setDeviceKey` function, and all other functions and contracts mentioned to improve overall system security and functionality.
- Document access control rules, system states, state transitions, and state machine diagrams to ensure consistency and proper validation checks.
- Regularly update and verify the integrity of dependencies.
- Use tools like Slither static analyzer and integrate it into the project's CI pipeline to improve code quality and catch common issues.
- Implement comprehensive unit testing, including positive, negative, and edge cases, to verify adherence to access controls and state transition specifications.

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 Manipulation	The justified use of inline assembly and low-level calls
Testing and Verification	The presence of robust testing procedures (e.g., unit tests, integration tests, and verification methods) and sufficient test coverage

Rating Criteria	
Rating	**Description**
Strong	No issues were found, and the system exceeds industry standards.
Satisfactory	Minor issues were found, but the system is compliant with best practices.
Moderate	Some issues that may affect system safety were found.
Weak	Many issues that affect system safety were found.
Missing	A required component is missing, significantly affecting system safety.
Not Applicable	The category is not applicable to this review.
Not Considered	The category was not considered in this review.
Further Investigation Required	Further investigation is required to reach a meaningful conclusion.

C. Code Quality Findings

This appendix lists findings that are not associated with specific vulnerabilities.

- **Do not allow immutable variables to be declared as constants.** Declaring immutable variables as constants can lead to confusion. Variables declared as constants in Solidity are understood to be compile-time constants, while immutable variables are not truly constant and can be assigned a value once during contract creation.
    - `FundTokenBase.sol#L28-L31`
- **Remove redundancies in constant values.** Redundant or overly complex names for constant values increase the risk of typographical errors, leading to potential bugs in the system. This could result in incorrect execution of the contract in an upgrade and possibly cause a loss of funds or other unintended consequences.
    - `FundTokenBase.sol#L34-L35`
    - `AuthorizationModule.sol#L16`
    - `TransactionalModule.sol#L31`
- **Eliminate code duplication.** Code duplication is problematic because it violates the don't repeat yourself (DRY) principle. It can lead to maintenance issues, increases the likelihood of introducing bugs during updates, and reduces overall code readability and efficiency.
    - `FundTokenBase.sol#L187-L191`
    - `FundTokenBase.sol#L231-L237`
- **Remove all unused code and contracts.** Leaving unused code and contracts in your smart contract can lead to confusion and misinterpretation of the contract's functionality. It also increases the complexity and size of the contract, potentially leading to higher gas costs and unnecessary vulnerabilities.
    - `AccessControlEnumerable` contract in `FundTokenBase` contract
- **Remove the virtual keyword on functions not meant to be overridden.** Having the virtual keyword on functions that are not meant to be overridden can lead to unintended behavior if a derived contract accidentally overrides them. This could result in bugs or, in the worst case, expose vulnerabilities in the smart contract. It can also cause misunderstandings about how the system works.

    - `AuthorizationModule.sol#L153`

- **Prevent contracts from existing in a state without vital role assignments.** Allowing a contract or system to exist in a state without a vital access role assigned can lead to unintended behavior.
    - `ROLE_WRITE_TRANSACTION` role in `MoneyMarketFund` contract

D. Automated Analysis Tool Configuration

Slither

We used Slither to detect common issues and anti-patterns in the codebase. Slither discovered a number of low- and medium-severity issues, such as TOB-FTMMF-05, TOB-FTMMF-07, and TOB-FTMMF-13. Integrating Slither into the project's testing environment can help find similar issues and improve the overall quality of the smart contracts' code.

```
slither --filter-paths mocks .
```

Figure D.1: The command used to run `slither-analyzer`

Integrating `slither-action` into the project's CI pipeline can automate this process.

Echidna

We used Echidna, a smart contract fuzzer that can rapidly test security properties via malicious, coverage-guided test case generation, to check various system states. Echidna uncovered issues such as TOB-FTMMF-04 and TOB-FTMMF-15. These issues would have otherwise been difficult to detect while relying solely on manual analysis.

```
function onlyShareholdersHaveBalance() public {
    assert(authorizationModule.getAuthorizedAccountsCount() >=
moneyMarketFund.getShareholdersWithHoldingsCount());
}
```

Figure D.2: An invariant implementation in the Echidna test harness

```
deployer: "0x30000"
sender: ["0x10000", "0x20000", "0x30000"]
corpusDir: "contracts/echidna/corpus"
testMode: "assertion"
testLimit: 500000
#coverage controls coverage guided testing
coverage: true
```

Figure D.3: Configuration setup in `config.yaml`

```
echidna . --contract MoneyMarketFundTest --config config.yaml
```

Figure D.4: The command used to run the Echidna fuzzing campaign

E. Delegatecall Proxy Guidance

The `delegatecall` opcode is a powerful feature that must be used carefully. Many high-profile exploits use little-known edge cases and counter-intuitive aspects of the `delegatecall` proxy pattern. This section outlines the most important risks to keep in mind while developing such smart contract systems. Trail of Bits developed the `slither-check-upgradability` tool to help develop secure `delegatecall` proxies; it performs safety checks relevant to both upgradeable and immutable `delegatecall` proxies.

- **Storage layout**. The storage layout of the proxy and implementation contracts must be the same. Do not try to define the same state variables on each contract. Instead, both contracts should inherit all their state variables from one shared base contract.
- **Inheritance**. If the base storage contract is split up, be aware that the order of inheritance impacts the final storage layout. For example, "`contract A is B,C`" and "`contract A is C,B`" will not yield the same storage layout if both `B` and `C` define state variables.
- **Initialization**. Make sure that the implementation is immediately initialized. Well-known disasters (and near disasters) have featured an uninitialized implementation contract. A factory pattern can help ensure that contracts are deployed and initialized correctly while also preventing front-running risks that might otherwise open up between contract deployment and initialization.
- **Function shadowing**. If the same method is defined in the proxy and the implementation, the proxy's function will not be called. Be aware of the `setOwner` and other administrative functions that commonly exist in proxies.
- **Preventing direct implementation usage**. Consider configuring the implementation's state variables with values that prevent the implementation from being used directly. For example, a flag could be set during construction that disables the implementation and causes all methods to revert. This is particularly important if the implementation also performs `delegatecall` operations because such operations could result in the unintended self-destruction of the implementation.
- **Immutable and constant variables**. These variables are embedded into a contract's bytecode and could, therefore, become out of sync between the proxy and the implementation. If the implementation has an incorrect immutable variable, this value may still be used even if the same variables are correctly set in the proxy's bytecode.

- **Contract existence checks**. All low-level calls, not just `delegatecall`, will return `true` on an address with empty bytecode. This return value could be misleading: the caller may interpret this return value to mean that the call successfully executed the operation when it did not. This return value may also cause important safety checks to be silently skipped. Be aware that while a contract's constructor is running, the contract address's bytecode remains empty until the end of the constructor's execution. We recommend rigorously verifying that all low-level calls are properly protected against nonexistent contracts. Keep in mind that most proxy libraries (such as the one written by OpenZeppelin) do not perform contract existence checks automatically.

For more information regarding `delegatecall` proxies, refer to our blog posts and presentations on the subject:

- Contract upgrade anti-patterns: This blog post describes the difference between a `delegatecall` proxy that uses a downstream data contract and one that uses an upstream data contract and how these patterns impact upgradeability.
- Good idea, bad design: How the Diamond standard falls short: This blog post dives deep into `delegatecall` risks that apply to all contracts, not only those that follow the Diamond standard.
- Breaking Aave Upgradeability: This blog post describes a subtle problem that Trail of Bits discovered in Aave `aToken` contracts, which resulted from the interplay between `delegatecall` proxies, contact existence checks, and unsafe initializations.
- Contract upgrade risks and recommendations: This Trail of Bits presentation describes best practices for developing upgradeable `delegatecall` proxies. The section starting at 5:49 describes some general risks that also apply to non-upgradeable proxies.

F. Security Best Practices for Using a Multisignature Owner Contract

Consensus requirements for sensitive actions, such as adjusting the balance of a user, are meant to mitigate the risks of the following cases:

- One person's judgment overrules the others'.
- One person's mistake causes a failure.
- One person's credentials are compromised, causing a failure.

In a 2-of-3 multisignature owner contract, for example, the execution of an admin-only transaction requires the consensus of two individuals in possession of two of the wallet's three private keys. For this model to be useful, it must fulfill the following requirements:

1. The private keys must be stored or held separately, and access to each one must be limited to a unique individual.
2. If the keys are physically held by third-party custodians (e.g., a bank), multiple keys should not be stored with the same custodian. (Doing so would violate requirement #1.)
3. The person asked to provide the second and final signature on a transaction (i.e., the co-signer) ought to refer to a pre-established policy specifying the conditions for approving the transaction by signing it with his or her key.
4. The co-signer also ought to verify that the half-signed transaction was generated willingly by the intended holder of the first signature's key.

Requirement #3 prevents the co-signer from becoming merely a "deputy" acting on behalf of the first signer (forfeiting the decision-making responsibility to the first signer and defeating the security model). If the co-signer can refuse to approve the transaction for any reason, the due-diligence conditions for approval may be unclear. That is why a policy for validating transactions is needed. A verification policy could include the following:

- A protocol for handling a request to co-sign a transaction (e.g., a half-signed transaction will be accepted only via an approved channel)
- An allowlist of specific Ethereum addresses allowed to be the payee of a transaction
- A limit on the number of actions in a day

Requirement #4 mitigates the risks associated with a single stolen key. For example, say that an attacker somehow acquired the unlocked Ledger Nano S of one of the signatories. A voice call from the co-signer to the initiating signatory to confirm the transaction would reveal that the key had been stolen and that the transaction should not be co-signed. If the signatory were under an active threat of violence, he or she could use a duress code (a code word, a phrase, or another signal agreed upon in advance) to covertly alert the others that the transaction had not been initiated willingly, without alerting the attacker.

G. Rounding Recommendations

The money market fund's arithmetic rounds down on every operation, which can lead to a loss of precision that benefits the user instead of the system. This was the root cause of a reported issue (TOB-FTMMF-15).

The following guidelines describe how to determine the rounding direction per operation. We recommend applying the same analysis for all arithmetic operations.

Determining Rounding Direction

To determine how to apply rounding (whether up or down), consider the result of the expected output.

For example, consider the formula for calculating the number of shares for the provided amount and price:

```
amount * NUMBER_SCALE_FACTOR / price
```

To benefit the fund, the rounding direction must tend towards lower values to minimize the number of shares users hold and can withdraw. Therefore it should round down.

Similar rounding techniques can be applied to all the system's formulas to ensure that rounding always occurs in the direction that benefits Franklin Templeton.

Rounding Rules

- When funds leave the system, these values **should round down** to favor the protocol over the user. Rounding these values up can allow attackers to profit from the rounding direction by receiving more than intended on fund interactions.
- When funds enter the system, these values **should always round up** to maximize the number of tokens a fund receives. Rounding these values down can result in near-zero values, which can allow attackers to profit from the rounding direction by receiving heavily discounted funds. Rounding down to zero can allow attackers to steal funds.
- If the result of computation can be positive or negative, then the **rounding direction must mirror the sign of the result** to benefit the fund.

Recommendations for Further Investigation

- Create unit tests to highlight the result of the precision loss. Unit tests will help validate the manual analysis.

- Use fuzzing and differential testing to find rounding issues.

H. Incident Response Recommendations

In this section, we provide recommendations around the formulation of an incident response plan.

- **Identify who (either specific people or roles) is responsible for carrying out the mitigations (deploying smart contracts, pausing contracts, upgrading the front end, etc.).**
    - Specifying these roles will strengthen the incident response plan and ease the execution of mitigating actions when necessary.
- **Document internal processes for situations in which a deployed remediation does not work or introduces a new bug.**
    - Consider adding a fallback scenario that describes an action plan in the event of a failed remediation.
- **Clearly describe the intended process of contract deployment.**
- **Consider whether and under what circumstances Chainflip will make affected users whole after certain issues occur.**
    - Some scenarios to consider include an individual or aggregate loss, a loss resulting from user error, a contract flaw, and a third-party contract flaw.
- **Document how Chainflip plans to keep up to date on new issues, both to inform future development and to secure the deployment toolchain and the external on-chain and off-chain services that the system relies on.**
    - For each language and component, describe the noteworthy sources for vulnerability news. Subscribe to updates for each source. Consider creating a special private Discord or Slack channel with a bot that will post the latest vulnerability news; this will help the team keep track of updates all in one place. Also consider assigning specific team members to keep track of the vulnerability news of a specific component of the system.
- **Consider scenarios involving issues that would indirectly affect the system.**
- **Determine when and how the team would reach out to and onboard external parties (auditors, affected users, other protocol developers, etc.).**
    - Some issues may require collaboration with external parties to efficiently remediate them.

- **Define behavior that is considered abnormal for off-chain monitoring.**
    - Consider adding more resilient solutions for detection and mitigation, especially in terms of specific alternate endpoints and queries for different data as well as status pages and support contacts for affected services.
- **Combine issues and determine whether new detection and mitigation scenarios are needed.**
- **Perform periodic dry runs of specific scenarios in the incident response plan to find gaps and opportunities for improvement and to develop muscle memory.**
    - Document the intervals at which the team should perform dry runs of the various scenarios. For scenarios that are more likely to happen, perform dry runs more regularly. Create a template to be filled in after a dry run to describe the improvements that need to be made to the incident response.

I. Fix Review Results

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

From September 5 to September 7, 2023, Trail of Bits reviewed the fixes and mitigations implemented by the Franklin Templeton team for the issues identified in this report. We reviewed each fix to determine its effectiveness in resolving the associated issue.

In summary, of the 15 issues described in this report, Franklin Templeton has resolved nine issues and has not resolved the remaining six issues. For additional information, please see the Detailed Fix Review Results below.

ID	Title	Status
1	Canceling all transaction requests causes DoS on MMF system	Resolved
2	Lack of validation in the IntentValidationModule contract can lead to inconsistent state	Resolved
3	Pending transactions cannot be settled	Resolved
4	Deauthorized accounts can keep shares of the MMF	Resolved
5	Solidity compiler optimizations can be problematic	Unresolved
6	Project dependencies contain vulnerabilities	Unresolved
7	Unimplemented getVersion function returns default value of zero	Resolved
8	The MultiSigGenVerifier threshold can be passed with a single signature	Resolved
9	Shareholders can renounce their authorization role	Resolved

10	Risk of multiple dividend payouts in a day	**Unresolved**
11	Shareholders can stop admin from deauthorizing them	**Unresolved**
12	Total number of submitters in MultiSigGenVerifier contract can be more than allowed limit of MAX_SUBMITTERS	**Resolved**
13	Lack of contract existence check on target address	**Resolved**
14	Pending transactions can trigger a DoS	**Unresolved**
15	Dividend distribution has an incorrect rounding direction for negative rates	**Unresolved**

Detailed Fix Review Results

TOB-FTMMF-01: Canceling all transaction requests causes DoS on MMF system
Resolved in commits `77a85b6` and `4e88ceb`. The Franklin Template team added a separate function, `cancelSelfServiceRequest`, to allow shareholders to cancel their requests and restricted the existing `cancelRequest` function in the `TransactionalModule` contract to be called only by the admin.

TOB-FTMMF-02: Lack of validation in the IntentValidationModule contract can lead to inconsistent state
Resolved in commit `094650d`. The Franklin Template team updated the `deviceKeyMap` variable to map keys to a pair of the account and device ID with the type `mapping(address => mapping(uint256 => string))`, which solves the key overriding issue for accounts with the same device ID.

TOB-FTMMF-03: Pending transactions cannot be settled
Resolved in commit `77a85b6`. The Franklin Template team added a check before removing the account from the `accountsWithTransactions` variable to ensure that the account does not have any pending transactions and also removed the `TransferAgentGateway` contract to simplify the system's access control rules.

TOB-FTMMF-04: Deauthorized accounts can keep shares of the MMF
Resolved in commit `cb2597c`. The Franklin Template team fixed the issue by adding a balance check before deauthorizing an account.

TOB-FTMMF-05: Solidity compiler optimizations can be problematic
Unresolved. The Franklin Template team provided the following context for this finding's fix status:

> *Optimizations are needed in order to deploy 3 of our contracts: MoneyMarketFund, TransactionalModule, and TransferAgentModule. Without those optimizations, the code cannot be deployed to mainnet due to its size; also, the gas savings with code optimization enabled are in the range of 10-25%. Additionally to this, the risks of bugs are low for the Solidity version we are using (0.8.18); the bugs reported are only exploitable for custom optimizations defined by the programmer and the use of legacy code generation, which we are not using at all.*

TOB-FTMMF-06: Project dependencies contain vulnerabilities
Unresolved. The Franklin Template team provided the following context for this finding's fix status:

> *The finding is identified by executing the npm audit. As confirmed by HardHat Toolbox maintainers on Fixing vulnerabilities in HardHat Toolbox · NomicFoundation/hardhat ·*

Discussion #3945 (github.com) and explained in this article, npm audit: Broken by Design — Overreacted, npm audit could generate false-positive results.

Currently, we are using the latest versions available for those dependencies, so those findings cannot be considered as vulnerabilities required to be fixed.

TOB-FTMMF-07: Unimplemented getVersion function returns default value of zero
Resolved in commit `9781fab`. The getVersion function of the `TransferAgentModule` contract now returns 1.

TOB-FTMMF-08: The MultiSigGenVerifier threshold can be passed with a single signature
Resolved in commit `f89e5f6`. The `signedDataExecution` function in the `MultiSigGenVerifier` contract now verifies the strictly ascending order of the recovered addresses to prevent the reuse of a signature multiple times.

TOB-FTMMF-09: Shareholders can renounce their authorization role
Resolved in commit `b177f55`. The Franklin Templeton team has added the `renounceRole` function in the `AuthorizationModule` contract, overriding the default functionality to prevent shareholders from deauthorizing themselves.

TOB-FTMMF-10: Risk of multiple dividend payouts in a day
Unresolved. The Franklin Template team provided the following context for this finding's fix status:

The validation is performed off-chain, and the dividend will not be posted multiple times unless it's intentional (i.e., it's being restated). Because there is a possibility of restating the dividend (or paying an adjusted div), for now we are leaving this logic on the client side, especially since we are the only ones that can invoke dividend distribution in the first place.

TOB-FTMMF-11: Shareholders can stop admin from deauthorizing them
Unresolved. The resolution for this has been postponed till The Franklin Template team allows self-custody of the funds. Users will not be able to execute this attack without having custody of the private key for their shareholder account.

The Franklin Template team provided the following context for this finding's fix status:

On hold until self-custody is implemented.

TOB-FTMMF-12: Total number of submitters in MultiSigGenVerifier contract can be more than allowed limit of MAX_SUBMITTERS
Resolved in commit `8cf3a3e`. The Franklin Templeton team has added a check to the `addSubmitters` function in the `MultiSigGenVerifier` contract to ensure that the total number of submitters cannot be more than the `MAX_SUBMITTERS`.

TOB-FTMMF-13: Lack of contract existence check on target address
Resolved in commit `a1a4534`. The issue has been resolved by adding a contract existence check to the `signedDataExecution` function in the `MultiSigGenVerifier` contract.

TOB-FTMMF-14: Pending transactions can trigger a DoS
Unresolved. The resolution for this has been postponed till The Franklin Template team allows self-custody of the funds. Users will not be able to create pending transactions without having custody of the private key for their shareholder account.

The Franklin Template team provided the following context for this finding's fix status:

> *On hold until self-custody is implemented.*

TOB-FTMMF-15: Dividend distribution has an incorrect rounding direction for negative rates
Unresolved. The Franklin Template team provided the following context for this finding's fix status:

> *This is a very unlikely scenario, but in any case, we would like to continue to accept a negative rate. The slight loss in precision during dividend distribution calculation that benefits the shareholders has been approved for now because a dividend rate means a current loss for the shareholder and because we also have mechanisms to adjust balances for shareholders.*

A. Fix Review Status Categories

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

Fix Status	
Status	**Description**
Undetermined	The status of the issue was not determined during this engagement.
Unresolved	The issue persists and has not been resolved.
Partially Resolved	The issue persists but has been partially resolved.
Resolved	The issue has been sufficiently resolved.