Fuji Protocol

Security Assessment

October 25, 2021

Prepared for:

Daigaro Cota
Fuji Protocol

Edgar Moreau
Fuji Protocol

Boyan Barakovu
Fuji Protocol

Prepared by:

Maximilian Krüger and Devashish Tomar

About Trail of Bits

Founded in 2012 and headquartered in New York, Trail of Bits provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code. With 80+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

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

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

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

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

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

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

Notices and Remarks

Classification and Copyright

This report is considered by Trail of Bits to be public information; it is licensed to Fuji Protocol under the terms of the project statement of work and has been made public at Fuji Protocol's request. Material within this report may not be reproduced or distributed in part or in whole without the express written permission of Trail of Bits.

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 mutually 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 such, 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.

Table of Contents

About Trail of Bits **1**

Notices and Remarks **2**

Table of Contents **2**

Project Summary **9**

Project Targets **10**

Project Coverage **11**

Codebase Maturity Evaluation **12**

Summary of Findings **15**

Detailed Findings **17**

1. Anyone can destroy the FujiVault logic contract if its initialize function was not called during deployment 17

2. Providers are implemented with delegatecall 19

3. Lack of contract existence check on delegatecall will result in unexpected 20

4. FujiVault.setFactor is unnecessarily complex and does not properly handle invalid input 23

5. Preconditions specified in docstrings are not checked by functions 25

6. The FujiERC1155.burnBatch function implementation is incorrect 27

7. Error in the white paper's equation for the cost of refinancing 29

8. Errors in the white paper's equation for index calculation 30

9. FujiERC1155.setURI does not adhere to the EIP-1155 specification 32

10. Partial refinancing operations can break the protocol 33

11. Native support for ether increases the codebase's complexity 34

12. Missing events for critical operations 35

13. Indexes are not updated before all operations that require up-to-date indexes 36

14. No protection against missing index updates before operations that depend on up-to-date indexes 37

15. Formula for index calculation is unnecessarily complex 38

16. Flasher’s initiateFlashloan function does not revert on invalid flashnum values 40

17. Docstrings do not reflect functions’ implementations 41

18. Harvester’s getHarvestTransaction function does not revert on invalid _farmProtocolNum and harvestType values 42

19. Lack of data validation in Controller’s doRefinancing function 44

20. Lack of data validation on function parameters 45

21. Solidity compiler optimizations can be problematic 46

A. Vulnerability Categories 47

B. Code Maturity Categories 49

C. Token Integration Checklist 51

E. Code Quality Recommendations 54

F. Index Construction for Interest Calculation 56

G. Handling Key Material 58

H. Fix Log 59

Overview

Fuji Protocol engaged Trail of Bits to review the security of its smart contracts. From October 4 to October 22, 2021, a team of two consultants conducted a security review of the client-provided source code, with six person-weeks of effort. Details of the project's timeline, test targets, and coverage are provided in subsequent sections of this report.

Project Scope

We focused our testing efforts on the identification of flaws that could result in a compromise or lapse of confidentiality, integrity, or availability of the target system. We performed automated testing and a manual review of the code, in addition to running system elements.

Summary of Findings

Our review resulted in four high-severity, two medium-severity, six low-severity, and nine informational-severity issues.

EXPOSURE ANALYSIS

Severity	*Count*
High	4
Medium	2
Low	6
Informational	9
Undetermined	0

CATEGORY BREAKDOWN

Category	*Count*
Denial of service	1
Data Validation	6
Arithmetic	4
Auditing and Logging	2
Undefined Behavior	8

Project Summary

Contact Information

The following managers were associated with this project:

Dan Guido, Account Manager
dan.guido@trailofbits.com

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

The following engineers were associated with this project:

Maximilian Krüger, Consultant
max.kruger@trailofbits.com

Devashish Tomar, Consultant
devashish.tomar@trailofbits.com

Project Timeline

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

Date	Event
September 30, 2021	Project pre-kickoff call
October 12, 2021	Status update meeting #1
October 18, 2021	Status update meeting #2
October 25, 2021	Delivery of report draft
October 25, 2021	Report readout meeting
December 3, 2021	Fix Log added (Appendix H)

Project Targets

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

Fuji Protocol

Repository	https://github.com/Fujicracy/fuji-protocol
Version	`933ea57b11889d87744efa23e95c90b7bf589402`
Type	Solidity
Platform	Ethereum

Project Coverage

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

FujiVault. Users interact with the Fuji Protocol through the FujiVault contract. With this contract, users can deposit and withdraw collateral and borrow and repay loans. For each collateral and borrowed asset pair, a FujiVault contract is deployed.

Controller. By calling doRefinance on the Controller contract, executors can start refinancing operations, which move collateral and debt to another borrowing protocol.

FLiquidator. Through the FLiquidator contract, users can liquidate undercollateralized positions within the Fuji Protocol.

FujiBaseERC1155. This contract is a base class for the FujiERC1155 contract and is responsible for implementing the core ERC-1155 functionality. We checked that the ERC-1155 specification is correctly implemented.

FujiERC1155. This contract is the main accounting system of the Fuji Protocol. It keeps track of the debt and collateral balances for each user on each vault. We focused on the implementation of the index functionality, which is further described in Appendix F.

F1155Manager. This contract is a base class for the FujiERC1155 contract. It allows the owner to grant permission to other addresses to call permissioned functions on the contract. We checked the correctness of the permissioning functionality.

Flasher. This contract is responsible for taking out flash loans from Aave, DyDx, and Cream. The loans are then used to move debt and collateral from one provider to another.

VaultControlUpgradeable. This contract is a base class for FujiVault. It allows admins to pause and resume the FujiVault contract. We checked the correctness of the pausing functionality.

VaultBaseUpgradeable. This contract is a base class for FujiVault. It provides functions that delegate calls to the given provider.

FujiAdmin. The FujiAdmin contract allows the owner to set a number of global addresses, such as the flasher, liquidator, and controller. These addresses are used by other contracts, such as the Flasher. The contract also maintains a whitelist of allowed vaults. We mainly focused on checking the correctness of the access controls.

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
Access Controls	We found no issues related to access controls. However, the protocol depends heavily on privileged operations, and there are no tests that highlight every access scenario.	**Moderate**
Arithmetic	The codebase consistently uses Solidity 0.8's safe math throughout. We found two errors in the arithmetic described in the Fuji Protocol white paper (TOB-FUJI-007, TOB-FUJI-008). We also found a high-severity issue related to the use of unscaled values in certain index calculations (TOB-FUJI-006).	**Moderate**
Assembly Use/Low-Level Calls	We found only one instance of assembly. However, this instance is complex and not sufficiently documented. The use of low-level operations is also limited. We found an unnecessary use of `delegatecall` to call providers, which resulted in a high-severity finding (TOB-FUJI-001).	**Moderate**
Code Stability	The code underwent frequent changes before the audit but was stable during the audit. However, there are leftover to-do comments throughout the codebase.	**Moderate**
Decentralization	Currently, a small number of admin externally owned accounts (EOAs) owned by the Fuji Protocol team have nearly total control over the protocol. This results in a centralized system, requiring users to trust a single entity.	**Weak**

Upgradeability	The Fuji Protocol team generally follows standard practices in using OpenZeppelin's upgradeable contracts, and the system uses the `hardhat-upgrades` plug-in. However, the use of `delegatecall` from assembly within a logic contract to explicitly work around `hardhat-upgrades`'s protections resulted in one high-severity finding (TOB-FUJI-001).	**Moderate**
Function Composition	The code is reasonably well structured. The functions have clear, narrow purposes, and critical functions can be easily extracted for testing.	**Satisfactory**
Front-Running	We did not find any issues related to front-running. However, we did not exhaustively check the protocol for front-running opportunities.	**Further investigation required**
Key Management	Admin accounts have permission to upgrade nearly all contracts and aspects of the protocol. The Fuji Protocol team informed us that the admin accounts are multisignature wallets whose signing keys are stored in hot wallets owned by Fuji Protocol team members.	**Moderate**
Monitoring	Many critical administrative functions do not emit events on important state changes (TOB-FUJI-012), making off-chain monitoring difficult to conduct. Additionally, the Fuji Protocol team informed us that they do not have an incident response plan and that off-chain components like Tenderly or Defender are currently not used to monitor on-chain activity.	**Weak**
Specification	The white paper is readable and provides a good overview of the protocol. However, we found two incorrect equations in the white paper. Additionally, the liquidation process of the protocol is underspecified: the white paper provides only a brief explanation of liquidation. While some parts of the codebase have appropriate inline comments, other parts are missing them. Moreover, some docstrings do not match their corresponding functions' implementations.	**Moderate**

Testing and Verification	The codebase contains integration tests for the most common operations. Unit tests and continuous integration are missing entirely.	Weak

Summary of Findings

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

ID	Title	Type	Severity
1	Anyone can destroy the FujiVault logic contract if its initialize function was not called during deployment	Denial of Service	**High**
2	Providers are implemented with delegatecall	Undefined Behavior	**Informational**
3	Lack of contract existence check on delegatecall will result in unexpected behavior	Data Validation	**High**
4	FujiVault.setFactor is unnecessarily complex and does not properly handle invalid input	Undefined Behavior	**Informational**
5	Preconditions specified in docstrings are not checked by functions	Data Validation	**Informational**
6	The FujiERC1155.burnBatch function implementation is incorrect	Arithmetic	**High**
7	Error in the white paper's equation for the cost of refinancing	Arithmetic	**Informational**
8	Errors in the white paper's equation for index calculation	Arithmetic	**Medium**
9	FujiERC1155.setURI does not adhere to the EIP-1155 specification	Auditing and Logging	**Informational**
10	Partial refinancing operations can break the protocol	Undefined Behavior	**Medium**
11	Native support for ether increases the codebase's complexity	Undefined Behavior	**Informational**

12	Missing events for critical operations	Auditing and Logging	**Low**
13	Indexes are not updated before all operations that require up-to-date indexes	Undefined Behavior	**High**
14	No protection against missing index updates before operations that depend on up-to-date indexes	Undefined Behavior	**Informational**
15	Formula for index calculation is unnecessarily complex	Arithmetic	**Informational**
16	Flasher’s initiateFlashloan function does not revert on invalid flashnum values	Data Validation	**Low**
17	Docstrings do not reflect functions’ implementations	Undefined Behavior	**Low**
18	Harvester’s getHarvestTransaction function does not revert on invalid _farmProtocolNum and harvestType values	Data Validation	**Low**
19	Lack of data validation in Controller’s doRefinancing function	Data Validation	**Low**
20	Lack of data validation on function parameters	Data Validation	**Low**
21	Solidity compiler optimizations can be problematic	Undefined Behavior	**Informational**

Detailed Findings

1. Anyone can destroy the FujiVault logic contract if its initialize function was not called during deployment	
Severity: **High**	Difficulty: **Medium**
Type: Denial of Service	Finding ID: TOB-FUJI-001
Target: `FujiVault.sol`	

Description

Anyone can destroy the `FujiVault` logic contract if its `initialize` function has not already been called. Calling `initialize` on a logic contract is uncommon, as usually nothing is gained by doing so. The deployment script does not call `initialize` on any logic contract. As a result, the exploit scenario detailed below is possible after deployment.

This issue is similar to a bug in AAVE that Trail of Bits found in 2020.

OpenZeppelin's `hardhat-upgrades` plug-in protects against this issue by disallowing the use of `selfdestruct` or `delegatecall` on logic contracts. However, the Fuji Protocol team has explicitly worked around these protections by calling `delegatecall` in assembly, which the plug-in does not detect.

Exploit Scenario

The Fuji contracts are deployed, but the `initialize` functions of the logic contracts are not called.

Bob, an attacker, deploys a contract to the address `alwaysSelfdestructs`, which simply always executes the `selfdestruct` opcode. Additionally, Bob deploys a contract to the address `alwaysSucceeds`, which simply never reverts.

Bob calls `initialize` on the `FujiVault` logic contract, thereby becoming its owner. To make the call succeed, Bob passes `0xEeeeeEeeeEeEeeEeEeEeeEEEeeeeEeeeeeeeEEeE` as the value for the `_collateralAsset` and `_borrowAsset` parameters. He then calls `FujiVaultLogic.setActiveProvider(alwaysSelfdestructs)`, followed by `FujiVault.setFujiERC1155(alwaysSucceeds)` to prevent an additional revert in the next and final call. Finally, Bob calls `FujiVault.deposit(1)`, sending 1 wei. This triggers a `delegatecall` to `alwaysSelfdestructs`, thereby destroying the `FujiVault` logic contract and making the protocol unusable until its proxy contract is upgraded.

Because OpenZeppelin's upgradeable contracts do not check for a contract's existence before a `delegatecall` (TOB-FUJI-003), all calls to the `FujiVault` proxy contract now succeed. This leads to exploits in any protocol integrating the Fuji Protocol. For example, a call that should repay all debt will now succeed even if no debt is repaid.

Recommendations

Short term, do not use `delegatecall` to implement providers. See TOB-FUJI-002 for more information.

Long term, avoid the use of `delegatecall`, as it is difficult to use correctly and can introduce vulnerabilities that are hard to detect.

2. Providers are implemented with delegatecall

Severity: **Informational**	Difficulty: **Undetermined**
Type: Undefined Behavior	Finding ID: TOB-FUJI-002
Target: `FujiVault.sol`, providers	

Description

The system uses `delegatecall` to execute an active provider's code on a `FujiVault`, making the `FujiVault` the holder of the positions in the borrowing protocol. However, `delegatecall` is generally error-prone, and the use of it introduced the high-severity finding TOB-FUJI-001.

It is possible to make a `FujiVault` the holder of the positions in a borrowing protocol without using `delegatecall`. Most borrowing protocols include a parameter that specifies the receiver of tokens that represent a position. For borrowing protocols that do not include this type of parameter, tokens can be transferred to the `FujiVault` explicitly after they are received from the borrowing protocol; additionally, the tokens can be transferred from the `FujiVault` to the provider before they are sent to the borrowing protocol. These solutions are conceptually simpler than and preferred to the current solution.

Recommendations

Short term, implement providers without the use of `delegatecall`. Set the receiver parameters to the `FujiVault`, or transfer the tokens corresponding to the position to the `FujiVault`.

Long term, avoid the use of `delegatecall`, as it is difficult to use correctly and can introduce vulnerabilities that are hard to detect.

3. Lack of contract existence check on delegatecall will result in unexpected behavior

Severity: **High**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-FUJI-003
Target: `VaultControlUpgradeable.sol`, `Proxy.sol`	

Description

The `VaultControlUpgradeable` and `Proxy` contracts use the `delegatecall` proxy pattern. If the implementation contract is incorrectly set or self-destructed, the contract may not be able to detect failed executions.

The `VaultControlUpgradeable` contract includes the `_execute` function, which users can invoke indirectly to execute a transaction to a `_target` address. This function does not check for contract existence before executing the `delegatecall` (figure 3.1).

```
/**
 * @dev Returns byte response of delegatcalls
 */
function _execute(address _target, bytes memory _data)
  internal
  whenNotPaused
  returns (bytes memory response)
{
  /* solhint-disable */
  assembly {
    let succeeded := delegatecall(sub(gas(), 5000), _target, add(_data, 0x20), mload(_data),
0, 0)
    let size := returndatasize()

    response := mload(0x40)
    mstore(0x40, add(response, and(add(add(size, 0x20), 0x1f), not(0x1f))))
    mstore(response, size)
    returndatacopy(add(response, 0x20), 0, size)

    switch iszero(succeeded)
    case 1 {
      // throw if delegatecall failed
      revert(add(response, 0x20), size)
    }
  }
  /* solhint-disable */
}
```

Figure 3.1: `fuji-protocol/contracts/abstracts/vault/VaultBaseUpgradeable.sol#L93-L115`

The `Proxy` contract, deployed by the `@openzeppelin/hardhat-upgrades` library, includes a payable fallback function that invokes the `_delegate` function when proxy calls are executed. This function is also missing a contract existence check (figure 3.2).

```
/**
 * @dev Delegates the current call to `implementation`.
 *
 * This function does not return to its internall call site, it will return directly to the
external caller.
 */
function _delegate(address implementation) internal virtual {
    // solhint-disable-next-line no-inline-assembly
    assembly {
        // Copy msg.data. We take full control of memory in this inline assembly
        // block because it will not return to Solidity code. We overwrite the
        // Solidity scratch pad at memory position 0.
        calldatacopy(0, 0, calldatasize())

        // Call the implementation.
        // out and outsize are 0 because we don't know the size yet.
        let result := delegatecall(gas(), implementation, 0, calldatasize(), 0, 0)

        // Copy the returned data.
        returndatacopy(0, 0, returndatasize())

        switch result
        // delegatecall returns 0 on error.
        case 0 { revert(0, returndatasize()) }
        default { return(0, returndatasize()) }
    }
}
```

Figure 3.2: `Proxy.sol#L16-L41`

A `delegatecall` to a destructed contract will return success (figure 3.3). Due to the lack of contract existence checks, a series of batched transactions may appear to be successful even if one of the transactions fails.

```
The low-level functions call, delegatecall and staticcall return true as their first
return value if the account called is non-existent, as part of the design of the
EVM. Account existence must be checked prior to calling if needed.
```

Figure 3.3: A snippet of the Solidity documentation detailing unexpected behavior related to `delegatecall`

Exploit Scenario

Eve upgrades the proxy to point to an incorrect new implementation. As a result, each `delegatecall` returns success without changing the state or executing code. Eve uses this to scam users.

Recommendations

Short term, implement a contract existence check before any `delegatecall`. Document the fact that `suicide` and `selfdestruct` can lead to unexpected behavior, and prevent future upgrades from using these functions.

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

References

- Contract Upgrade Anti-Patterns
- Breaking Aave Upgradeability

4. FujiVault.setFactor is unnecessarily complex and does not properly handle invalid input

Severity: **Informational**	Difficulty: **Undetermined**
Type: Undefined Behavior	Finding ID: TOB-FUJI-004
Target: `FujiVault.sol`	

Description

The `FujiVault` contract's `setFactor` function sets one of four state variables to a given value. Which state variable is set depends on the value of a string parameter. If an invalid value is passed, `setFactor` succeeds but does not set any of the state variables.

This creates edge cases, makes writing correct code more difficult, and increases the likelihood of bugs.

```
function setFactor(
    uint64 _newFactorA,
    uint64 _newFactorB,
    string calldata _type
) external isAuthorized {
    bytes32 typeHash = keccak256(abi.encode(_type));
    if (typeHash == keccak256(abi.encode("collatF"))) {
        collatF.a = _newFactorA;
        collatF.b = _newFactorB;
    } else if (typeHash == keccak256(abi.encode("safetyF"))) {
        safetyF.a = _newFactorA;
        safetyF.b = _newFactorB;
    } else if (typeHash == keccak256(abi.encode("bonusLiqF"))) {
        bonusLiqF.a = _newFactorA;
        bonusLiqF.b = _newFactorB;
    } else if (typeHash == keccak256(abi.encode("protocolFee"))) {
        protocolFee.a = _newFactorA;
        protocolFee.b = _newFactorB;
    }
}
```

Figure 4.1: `FujiVault.sol#L475-494`

Exploit Scenario

A developer on the Fuji Protocol team calls `setFactor` from another contract. He passes a type that is not handled by `setFactor`. As a result, code that is expected to set a state variable does nothing, resulting in a more severe vulnerability.

Recommendations

Short term, replace `setFactor` with four separate functions, each of which sets one of the four state variables.

Long term, avoid string constants that simulate enumerations, as they cannot be checked by the typechecker. Instead, use enums and ensure that any code that depends on enum values handles all possible values.

5. Preconditions specified in docstrings are not checked by functions

Severity: **Informational**	Difficulty: **Undetermined**
Type: Data Validation	Finding ID: TOB-FUJI-005
Target: `FujiVault.sol`, `Controller.sol`	

Description

The docstrings of several functions specify preconditions that the functions do not automatically check for.

For example, the docstring of the `FujiVault` contract's `setFactor` function contains the preconditions shown in figure 5.1, but the function's body does not contain the corresponding checks shown in figure 5.2.

```
* For safetyF;  Sets Safety Factor of Vault, should be > 1, a/b
* For collatF; Sets Collateral Factor of Vault, should be > 1, a/b
```

Figure 5.1: `FujiVault.sol#L469-470`

```
require(safetyF.a > safetyF.b);
...
require(collatF.a > collatF.b);
```

Figure 5.2: The checks that are missing from `FujiVault.setFactor`

Additionally, the docstring of the `Controller` contract's `doRefinancing` function contains the preconditions shown in figure 5.3, but the function's body does not contain the corresponding checks shown in figure 5.4.

```
* @param _ratioB: _ratioA/_ratioB <= 1, and > 0
```

Figure 5.3: `Controller.sol#L41`

```
require(ratioA > 0 && ratioB > 0);
require(ratioA <= ratioB);
```

Figure 5.4: The checks that are missing from `Controller.doRefinancing`

Exploit Scenario

The `setFactor` function is called with values that violate its documented preconditions. Because the function does not check for these preconditions, unexpected behavior occurs.

Recommendations

Short term, add checks for preconditions to all functions with preconditions specified in their docstrings.

Long term, ensure that all documentation and code are in sync.

6. The FujiERC1155.burnBatch function implementation is incorrect

Severity: **High**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-FUJI-006
Target: `FujiERC1155.sol`	

Description

The `FujiERC1155` contract's `burnBatch` function deducts the unscaled `amount` from the user's balance and from the total supply of an asset. If the liquidity index of an asset (`index[assetId]`) is different from its initialized value, the execution of `burnBatch` could result in unintended arithmetic calculations. Instead of deducting the `amount` value, the function should deduct the `amountScaled` value.

```
function burnBatch(
  address _account,
  uint256[] memory _ids,
  uint256[] memory _amounts
) external onlyPermit {
  require(_account != address(0), Errors.VL_ZERO_ADDR_1155);
  require(_ids.length == _amounts.length, Errors.VL_INPUT_ERROR);

  address operator = _msgSender();

  uint256 accountBalance;
  uint256 assetTotalBalance;
  uint256 amountScaled;

  for (uint256 i = 0; i < _ids.length; i++) {
    uint256 amount = _amounts[i];

    accountBalance = _balances[_ids[i]][_account];
    assetTotalBalance = _totalSupply[_ids[i]];

    amountScaled = _amounts[i].rayDiv(indexes[_ids[i]]);

    require(amountScaled != 0 && accountBalance >= amountScaled,
Errors.VL_INVALID_BURN_AMOUNT);

    _balances[_ids[i]][_account] = accountBalance - amount;
    _totalSupply[_ids[i]] = assetTotalBalance - amount;
  }

  emit TransferBatch(operator, _account, address(0), _ids, _amounts);
}
```

Figure 6.1: `FujiERC1155.sol#L218-247`

Exploit Scenario

The `burnBatch` function is called with an asset for which the liquidity index is different from its initialized value. Because `amount` was used instead of `amountScaled`, unexpected behavior occurs.

Recommendations

Short term, revise the `burnBatch` function so that it uses `amountScaled` instead of `amount` when updating a user's balance and the total supply of an asset.

Long term, use the `burn` function in the `burnBatch` function to keep functionality consistent.

7. Error in the white paper's equation for the cost of refinancing

Severity: **Informational**	Difficulty: **Undetermined**
Type: Arithmetic	Finding ID: TOB-FUJI-007
Target: White paper	

Description

The white paper uses the following equation (equation 4) to describe how the cost of refinancing is calculated:

$$R_{cost =} Tx_{gas} + G_{price} + ETH_{price} + B_{debt} + FL_{fee}$$

B_{debt} is the amount of debt to be refinanced and is a summand of the equation. This is incorrect, as it implies that the refinancing cost is always greater than the amount of debt to be refinanced.

A correct version of the equation could be $R_{cost =} Tx_{gas} + G_{price} + ETH_{price} + FL_{fee}$, in which FL_{fee} is an amount, or $R_{cost =} Tx_{gas} + G_{price} + ETH_{price} + B_{debt} * FL_{fee}$, in which FL_{fee} is a percentage.

Recommendations

Short term, fix equation 4 in the white paper.

Long term, ensure that the equations in the white paper are correct and in sync with the implementation.

8. Errors in the white paper’s equation for index calculation

Severity: **Medium**	Difficulty: **Undetermined**
Type: Arithmetic	Finding ID: TOB-FUJI-008
Target: White paper	

Description

The white paper uses the following equation (equation 1) to describe how the index for a given token at timestamp t is calculated:

$$I_t = I_{t-1} + (B_{t-1} - B_t)/B_{t-1}$$

B_t is the amount of the given token that the Fuji Protocol owes the provider (the borrowing protocol) at timestamp t.

The index is updated only when the balance changes through the accrual of interest, not when the balance changes through borrowing or repayment operations. This means that $B_{t-1} - B_t$ is always negative, which is incorrect, as $(B_{t-1} - B_t)/B_{t-1}$ should calculate the interest rate since the last index update.

The index represents the total interest rate since the deployment of the protocol. It is the product of the various interest rates accrued on the active providers during the lifetime of the protocol (measured only during state-changing interactions with the provider): $r_1 * r_2 * r_3 * ... * r_n$. A user's current balance is computed by taking the user’s initial stored balance, multiplying it by the current index, and dividing it by the index at the time of the creation of that user's position. The division operation ensures that the user will not owe interest that accrued before the creation of the user’s position. The index provides an efficient way to keep track of interest rates without having to update each user's balance separately, which would be prohibitively expensive on Ethereum.

However, interest is compounded through multiplication, not addition. The formula should use the product sign instead of the plus sign.

Exploit Scenario

Alice decides to use the Fuji Protocol after reading the white paper. She later learns that calculations in the white paper do not match the implementations in the protocol. Because Alice allocated her funds based on her understanding of the specification, she loses funds.

Recommendations

Short term, replace equation 1 in the white paper with a correct and simplified version. For more information on the simplified version, see finding TOB-FUJI-015.

$$I_t = I_{t-1} * B_t / B_{t-1}$$

Long term, ensure that the equations in the white paper are correct and in sync with the implementation.

9. FujiERC1155.setURI does not adhere to the EIP-1155 specification

Severity: **Informational**	Difficulty: **Undetermined**
Type: Auditing and Logging	Finding ID: TOB-FUJI-009
Target: `FujiERC1155.sol`	

Description

The `FujiERC1155` contract's `setURI` function does not emit the `URI` event.

```
/**
 * @dev Sets a new URI for all token types, by relying on the token type ID
 */
function setURI(string memory _newUri) public onlyOwner {
  _uri = _newUri;
}
```

Figure 9.1: `FujiERC1155.sol#L266-268`

This behavior does not adhere to the EIP-1155 specification, which states the following:

```
Changes to the URI MUST emit the URI event if the change can be expressed with an
event (i.e. it isn't dynamic/programmatic).
```

Figure 9.2: A snippet of the EIP-1155 specification

Recommendations

Short term, revise the `setURI` function so that it emits the `URI` event.

Long term, review the EIP-1155 specification to verify that the contracts adhere to the standard.

References

- EIP-1155

10. Partial refinancing operations can break the protocol

Severity: **Medium**	Difficulty: **Medium**
Type: Undefined Behavior	Finding ID: TOB-FUJI-010
Target: `FujiVault.sol`, `Controller.sol`, white paper	

Description

The white paper documents the `Controller` contract's ability to perform partial refinancing operations. These operations move only a fraction of debt and collateral from one provider to another to prevent unprofitable interest rate slippage.

However, the protocol does not correctly support partial refinancing situations in which debt and collateral are spread across multiple providers. For example, payback and withdrawal operations always interact with the current provider, which might not contain enough funds to execute these operations. Additionally, the interest rate indexes are computed only from the debt owed to the current provider, which might not accurately reflect the interest rate across all providers.

Exploit Scenario

An executor performs a partial refinancing operation. Interest rates are computed incorrectly, resulting in a loss of funds for either the users or the protocol.

Recommendations

Short term, disable partial refinancing until the protocol supports it in all situations.

Long term, ensure that functionality that is not fully supported by the protocol cannot be used by accident.

11. Native support for ether increases the codebase’s complexity	
Severity: **Informational**	Difficulty: **Undetermined**
Type: Undefined Behavior	Finding ID: TOB-FUJI-011
Target: Throughout	

Description

The protocol supports ERC20 tokens and Ethereum’s native currency, ether.

Ether transfers follow different semantics than token transfers. As a result, many functions contain extra code, like the code shown in figure 11.1, to handle ether transfers.

```
if (vAssets.borrowAsset == ETH) {
  require(msg.value >= amountToPayback, Errors.VL_AMOUNT_ERROR);
  if (msg.value > amountToPayback) {
    IERC20Upgradeable(vAssets.borrowAsset).univTransfer(
      payable(msg.sender),
      msg.value - amountToPayback
    );
  }
} else {
  // Check User Allowance
  require(
    IERC20Upgradeable(vAssets.borrowAsset).allowance(msg.sender, address(this)) >=
      amountToPayback,
    Errors.VL_MISSING_ERC20_ALLOWANCE
  );
```

Figure 11.1: `FujiVault.sol#L319-333`

This extra code increases the codebase’s complexity. Furthermore, functions will behave differently depending on their arguments.

Recommendations

Short term, replace native support for ether with support for ERC20 WETH. This will decrease the complexity of the protocol and the likelihood of bugs.

12. Missing events for critical operations

Severity: **Low**	Difficulty: **Low**
Type: Auditing and Logging	Finding ID: TOB-FUJI-012
Target: Throughout	

Description

Many functions that make important state changes do not emit events. These functions include, but are not limited to, the following:

- All setters in the `FujiAdmin` contract
- The `setFujiAdmin`, `setFujiERC1155`, `setFactor`, `setOracle`, and `setProviders` functions in the `FujiVault` contract
- The `setMapping` and `setURI` functions in the `FujiMapping` contract
- The `setFujiAdmin` and `setExecutors` functions in the `Controller` contract
- The `setURI` and `setPermit` functions in the `FujiERC1155` contract
- The `setPriceFeed` function in the `FujiOracle` contract

Exploit scenario

An attacker gains permission to execute an operation that changes critical protocol parameters. She executes the operation, which does not emit an event. Neither the Fuji Protocol team nor the users are notified about the parameter change. The attacker uses the changed parameter to steal funds. Later, the attack is detected due to the missing funds, but it is too late to react and mitigate the attack.

Recommendations

Short term, ensure that all state-changing operations emit events.

Long term, use an event monitoring system like Tenderly or Defender, use Defender’s automated incident response feature, and develop an incident response plan to follow in case of an emergency.

13. Indexes are not updated before all operations that require up-to-date indexes	
Severity: **High**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-FUJI-013
Target: `FujiVault.sol`, `FujiERC1155.sol`, `FLiquidator.sol`	

Description

The `FujiERC1155` contract uses indexes to keep track of interest rates. Refer to Appendix F for more detail on the index calculation.

The `FujiVault` contract's `updateF1155Balances` function is responsible for updating indexes. However, this function is not called before all operations that read indexes. As a result, these operations use outdated indexes, which results in incorrect accounting and could make the protocol vulnerable to exploits.

`FujiVault.deposit` calls `FujiERC1155._mint`, which reads indexes but does not call `updateF1155Balances`.

`FujiVault.paybackLiq` calls `FujiERC1155.balanceOf`, which reads indexes but does not call `updateF1155Balances`.

Exploit Scenario

The indexes have not been updated in one day. User Bob deposits collateral into the `FujiVault`. Day-old indexes are used to compute Bob's scaled amount, causing Bob to gain interest for an additional day for free.

Recommendations

Short term, ensure that all operations that require up-to-date indexes first call `updateF1155Balances`. Write tests for each function that depends on up-to-date indexes with assertions that fail if indexes are outdated.

Long term, redesign the way indexes are accessed and updated such that a developer cannot simply forget to call `updateF1155Balances`.

14. No protection against missing index updates before operations that depend on up-to-date indexes	
Severity: **Informational**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-FUJI-014
Target: `FujiVault.sol`, `FujiERC1155.sol`, `FLiquidator.sol`	

Description

The `FujiERC1155` contract uses indexes to keep track of interest rates. Refer to Appendix F for more detail on the index calculation.

The `FujiVault` contract's `updateF1155Balances` function is responsible for updating indexes. This function must be called before all operations that read indexes (TOB-FUJI-013). However, the protocol does not protect against situations in which indexes are not updated before they are read; these situations could result in incorrect accounting.

Exploit Scenario

Developer Bob adds a new operation that reads indexes, but he forgets to add a call to `updateF1155Balances`. As a result, the new operation uses outdated index values, which causes incorrect accounting.

Recommendations

Short term, redesign the index calculations so that they provide protection against the reading of outdated indexes.

For example, the index calculation process could keep track of the last index update's block number and access indexes exclusively through a getter, which updates the index automatically, if it has not already been updated for the current block. Since ERC-1155's `balanceOf` and `totalSupply` functions do not allow side effects, this solution would require the use of different functions internally.

Long term, use defensive coding practices to ensure that critical operations are always executed when required.

15. Formula for index calculation is unnecessarily complex

Severity: **Informational**	Difficulty: **Undetermined**
Type: Arithmetic	Finding ID: TOB-FUJI-015
Target: `FujiERC1155.sol`	

Description

Indexes are updated within the `FujiERC1155` contract's `updateState` function, shown in figure 15.1. Refer to Appendix F for more detail on the index calculation.

```
function updateState(uint256 _assetID, uint256 newBalance) external override onlyPermit {
  uint256 total = totalSupply(_assetID);

  if (newBalance > 0 && total > 0 && newBalance > total) {
    uint256 diff = newBalance - total;

    uint256 amountToIndexRatio = (diff.wadToRay()).rayDiv(total.wadToRay());

    uint256 result = amountToIndexRatio + WadRayMath.ray();

    result = result.rayMul(indexes[_assetID]);
    require(result <= type(uint128).max, Errors.VL_INDEX_OVERFLOW);

    indexes[_assetID] = uint128(result);

    // TODO: calculate interest rate for a fujiOptimizer Fee.
  }
}
```

Figure 15.1: `FujiERC1155.sol#L40-57`

The code in figure 14.1 translates to the following equation:

$$index_t = index_{t-1} * (1 + (balance_t - balance_{t-1})/balance_{t-1})$$

Using the distributive property, we can transform this equation into the following:

$$index_t = index_{t-1} * (1 + balance_t/balance_{t-1} - balance_{t-1}/balance_{t-1})$$

This version can then be simplified:

$$index_t = index_{t-1} * (1 + balance_t/balance_{t-1} - 1)$$

Finally, we can simplify the equation even further:

$$index_t = index_{t-1} * balance_t / balance_{t-1}$$

The resulting equation is simpler and more intuitively conveys the underlying idea—that the index grows by the same ratio as the balance grew since the last index update.

Recommendations

Short term, use the simpler index calculation formula in the `updateState` function of the `Fuji1155Contract`. This will result in code that is more intuitive and that executes using slightly less gas.

Long term, use simpler versions of the equations used by the protocol to make the arithmetic easier to understand and implement correctly.

16. Flasher’s initiateFlashloan function does not revert on invalid flashnum values

Severity: **Low**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-FUJI-016
Target: `Flasher.sol`	

Description

The `Flasher` contract’s `initiateFlashloan` function does not initiate a flash loan or perform a refinancing operation if the `flashnum` parameter is set to a value greater than 2. However, the function does not revert on invalid `flashnum` values.

```
function initiateFlashloan(FlashLoan.Info calldata info, uint8 _flashnum) external
isAuthorized {
   if (_flashnum == 0) {
     _initiateAaveFlashLoan(info);
   } else if (_flashnum == 1) {
     _initiateDyDxFlashLoan(info);
   } else if (_flashnum == 2) {
     _initiateCreamFlashLoan(info);
   }
 }
```

Figure 16.1: `Flasher.sol#L61-69`

Exploit Scenario

Alice, an executor of the Fuji Protocol, calls `Controller. doRefinancing` with the `flashnum` parameter set to 3. As a result, no flash loan is initialized, and no refinancing happens; only the active provider is changed. This results in unexpected behavior. For example, if a user wants to repay his debt after refinancing, the operation will fail, as no debt is owed to the active provider.

Recommendations

Short term, revise `initiateFlashloan` so that it reverts when it is called with an invalid `flashnum` value.

Long term, ensure that all functions revert if they are called with invalid values.

17. Docstrings do not reflect functions’ implementations

Severity: **Low**	Difficulty: **Undetermined**
Type: Undefined Behavior	Finding ID: TOB-FUJI-017
Target: `FujiVault.sol`	

Description

The docstring of the `FujiVault` contract’s `withdraw` function states the following:

```
* @param _withdrawAmount: amount of collateral to withdraw
* otherwise pass -1 to withdraw maximum amount possible of collateral (including safety
factors)
```

Figure 17.1: `FujiVault.sol#L188-189`

However, the maximum amount is withdrawn on any negative value, not only on a value of -1.

A similar inconsistency between the docstring and the implementation exists in the `FujiVault` contract’s `payback` function.

Recommendations

Short term, adjust the `withdraw` and `payback` functions’ docstrings or their implementations to make them match.

Long term, ensure that docstrings always match the corresponding function’s implementation.

18. Harvester’s getHarvestTransaction function does not revert on invalid _farmProtocolNum and harvestType values

Severity: **Low**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-FUJI-018
Target: `Harvester.sol`	

Description

The `Harvester` contract’s `getHarvestTransaction` function incorrectly returns `claimedToken` and `transaction` values of 0 if the `_farmProtocolNum` parameter is set to a value greater than 1 or if the `harvestType` value is set to value greater than 2. However, the function does not revert on invalid `_farmProtocolNum` and `harvestType` values.

```
function getHarvestTransaction(uint256 _farmProtocolNum, bytes memory _data)
  external
  view
  override
  returns (address claimedToken, Transaction memory transaction)
{
  if (_farmProtocolNum == 0) {
    transaction.to = 0x3d9819210A31b4961b30EF54bE2aeD79B9c9Cd3B;
    transaction.data = abi.encodeWithSelector(
      bytes4(keccak256("claimComp(address)")),
      msg.sender
    );
    claimedToken = 0xc00e94Cb662C3520282E6f5717214004A7f26888;
  } else if (_farmProtocolNum == 1) {
    uint256 harvestType = abi.decode(_data, (uint256));

    if (harvestType == 0) {
      // claim
      (, address[] memory assets) = abi.decode(_data, (uint256, address[]));
      transaction.to = 0xd784927Ff2f95ba542BfC824c8a8a98F3495f6b5;
      transaction.data = abi.encodeWithSelector(
        bytes4(keccak256("claimRewards(address[],uint256,address)")),
        assets,
        type(uint256).max,
        msg.sender
      );
    } else if (harvestType == 1) {
      //
      transaction.to = 0x4da27a545c0c5B758a6BA100e3a049001de870f5;
      transaction.data = abi.encodeWithSelector(bytes4(keccak256("cooldown()")));
    } else if (harvestType == 2) {
      //
      transaction.to = 0x4da27a545c0c5B758a6BA100e3a049001de870f5;
      transaction.data = abi.encodeWithSelector(
        bytes4(keccak256("redeem(address,uint256)")),
        msg.sender,
        type(uint256).max
      );
      claimedToken = 0x7Fc66500c84A76Ad7e9c93437bFc5Ac33E2DDaE9;
    }
  }
}
```

Figure 18.1: `Harvester.sol#L13-54`

Exploit Scenario

Alice, an executor of the Fuji Protocol, calls `getHarvestTransaction` with the `_farmProtocolNum` parameter set to 2. As a result, rather than reverting, the function returns `claimedToken` and `transaction` values of 0.

Recommendations

Short term, revise `getHarvestTransaction` so that it reverts if it is called with invalid `farmProtocolNum` or `harvestType` values.

Long term, ensure that all functions revert if they are called with invalid values.

19. Lack of data validation in Controller’s doRefinancing function

Severity: **Low**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-FUJI-019
Target: `Controller.sol`	

Description

The `Controller` contract’s `doRefinancing` function does not check the `_newProvider` value. Therefore, the function accepts invalid values for the `_newProvider` parameter.

```
function doRefinancing(
  address _vaultAddr,
  address _newProvider,
  uint256 _ratioA,
  uint256 _ratioB,
  uint8 _flashNum
) external isValidVault(_vaultAddr) onlyOwnerOrExecutor {
  IVault vault = IVault(_vaultAddr);
  [...]
  [...]
  IVault(_vaultAddr).setActiveProvider(_newProvider);
}
```

Figure 19.1: `Controller.sol#L44-84`

Exploit Scenario

Alice, an executor of the Fuji Protocol, calls `Controller.doRefinancing` with the `_newProvider` parameter set to the same address as the active provider. As a result, unnecessary flash loan fees will be paid.

Recommendations

Short term, revise the `doRefinancing` function so that it reverts if `_newProvider` is set to the same address as the active provider.

Long term, ensure that all functions revert if they are called with invalid values.

20. Lack of data validation on function parameters	
Severity: **Low**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-FUJI-020
Target: Throughout	

Description

Certain setter functions fail to validate the addresses they receive as input. The following addresses are not validated:

- The addresses passed to all setters in the `FujiAdmin` contract
- The `_newFujiAdmin` address in the `setFujiAdmin` function in the `Controller` and `FujiVault` contracts
- The `_provider` address in the `FujiVault.setActiveProvider` function
- The `_oracle` address in the `FujiVault.setOracle` function
- The `_providers` addresses in the `FujiVault.setProviders` function
- The `newOwner` address in the `transferOwnership` function in the `Claimable` and `ClaimableUpgradeable` contracts

Exploit scenario

Alice, a member of the Fuji Protocol team, invokes the `FujiVault.setOracle` function and sets the oracle address as `address(0)`. As a result, code relying on the oracle address is no longer functional.

Recommendations

Short term, add zero-value or contract existence checks to the functions listed above to ensure that users cannot accidentally set incorrect values, misconfiguring the protocol.

Long term, use Slither, which will catch missing zero checks.

21. Solidity compiler optimizations can be problematic

Severity: **Informational**	Difficulty: **High**
Type: Undefined Behavior	Finding ID: TOB-FUJI-021
Target: `hardhat-config.js`	

Description

Fuji Protocol has enabled optional compiler optimizations in Solidity.

There have been several optimization bugs with security implications. Moreover, optimizations are actively being developed. Solidity compiler optimizations are disabled by default, and it is unclear how many contracts in the wild actually use them. Therefore, it is unclear how well they are being tested and exercised.

High-severity security issues due to optimization bugs have occurred in the past. A high-severity bug in the `emscripten`-generated `solc-js` compiler used by Truffle and Remix persisted until late 2018. The fix for this bug was not reported in the Solidity CHANGELOG. Another high-severity optimization bug resulting in incorrect bit shift results was patched in Solidity 0.5.6. More recently, another bug due to the incorrect caching of `keccak256` was reported.

A compiler audit of Solidity from November 2018 concluded that the optional optimizations may not be safe.

It is likely that there are latent bugs related to optimization and that new bugs will be introduced due to future optimizations.

Exploit Scenario

A latent or future bug in Solidity compiler optimizations—or in the Emscripten transpilation to `solc-js`—causes a security vulnerability in the Fuji Protocol 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.

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 of users 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	Breach of the confidentiality or integrity of data
Data Exposure	Exposure of sensitive information
Data Validation	Improper reliance on the structure or values of data
Denial of Service	System failure with an availability impact
Error Reporting	Insecure or insufficient reporting of error conditions
Patching	Outdated software package or library
Session Management	Improper identification of authenticated users
Testing	Insufficient test methodology or test coverage
Timing	Race conditions, locking, 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 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 client 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 Categories

The following tables describe the code maturity categories and rating criteria used in this document.

Code Maturity Categories

Categories	Description
Access Controls	The authentication and authorization of components
Arithmetic	The proper use of mathematical operations and semantics
Assembly Use	The use of inline assembly
Centralization	The existence of a single point of failure
Upgradeability	Contract upgradeability
Function Composition	The separation of the logic into functions with clear purposes
Front-Running	Resistance to front-running
Key Management	The existence of proper procedures for key generation, distribution, and access
Monitoring	The use of events and monitoring procedures
Specification	The comprehensiveness and readability of codebase documentation and specification
Testing and Verification	The use of testing techniques (e.g., unit tests and fuzzing)

Rating Criteria

Rating	Description
Strong	The control was robust, documented, automated, and comprehensive.
Satisfactory	With a few minor exceptions, the control was applied consistently.
Moderate	The control was applied inconsistently in certain areas.
Weak	The control was applied inconsistently or not at all.
Missing	The control was missing.
Not Applicable	The control is not applicable.
Not Considered	The control was not reviewed.
Further Investigation Required	The control requires further investigation.

C. Token Integration Checklist

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

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

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

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

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

General Security Considerations

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

ERC Conformity

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

- ❏ **`Transfer` and `transferFrom` return a boolean.** Several tokens do not return a boolean on these functions. As a result, their calls in the contract might fail.
- ❏ **The `name`, `decimals`, and `symbol` functions are present if used.** These functions are optional in the ERC20 standard and may not be present.

- ❏ **`Decimals` returns a `uint8`.** Several tokens incorrectly return a `uint256`. In such cases, ensure that the value returned is below 255.
- ❏ **The token mitigates the known ERC20 race condition.** The ERC20 standard has a known ERC20 race condition that must be mitigated to prevent attackers from stealing tokens.
- ❏ **The token is not an ERC777 token and has no external function call in `transfer` or `transferFrom`.** External calls in the transfer functions can lead to reentrancies.

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

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

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

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

Contract Composition

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

Owner Privileges

- ❏ **The token is not upgradeable.** Upgradeable contracts may change their rules over time. Use Slither's `human-summary` printer to determine if the contract is upgradeable.
- ❏ **The owner has limited minting capabilities.** Malicious or compromised owners can abuse minting capabilities. Use Slither's `human-summary` printer to review minting capabilities, and consider manually reviewing the code.
- ❏ **The token is not pausable.** Malicious or compromised owners can trap contracts relying on pausable tokens. Identify pausable code by hand.
- ❏ **The owner cannot blacklist the contract.** Malicious or compromised owners can trap contracts relying on tokens with a blacklist. Identify blacklisting features by hand.
- ❏ **The team behind the token is known and can be held responsible for abuse.** Contracts with anonymous development teams or teams that reside in legal shelters require a higher standard of review.

Token Scarcity

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

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

E. Code Quality Recommendations

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

General Recommendations

- Consider merging `FujiVault`, `VaultBaseUpgradeable`, and `VaultControlUpgradeable`. These separate contracts are unnecessary and make auditing more difficult.
- Consider removing unused imports from the `DyDxFlashLoans`, `FujiERC115`, `Harvester`, and `Swapper` contracts.

`Flasher`

- Consider replacing the magic numbers for the `flashnum` parameter of the `initiateFlashloan` function with enums.

`FujiVault`

- The fee calculation implemented by `_userProtocolFee` is repeated inline in the `borrow` function (line 275). Consider calling `_userProtocolFee` instead of `borrow`.
- Consider renaming the `_fujiadmin` parameter of the `initialize` function to `__fujiAdmin`. To prevent shadowing, do not use entirely lowercase names.
- Consider renaming the `Factor` struct to `Ratio`, its field `a` to `numerator`, and its field `b` to `denominator`; these labels are more self-explanatory.
- Consider changing the type of `fujiERC1155` from `address` to `IFujiERC1155`; this will reduce boilerplate casts, which decrease readability.
- Both the `if` and `else` branch of the `deposit` function revert if `_collateralAmount == 0`. Consider moving that check to before the `if` statement.
- Consider renaming `updateF1155Balances` to `updateF1155Indexes`, which more accurately describes what the function does.

`FujiBaseERC1155`

- Consider removing the unused `_beforeTokenTransfer` and `_asSingletonArray` functions.

`FujiERC1155`

- Consider renaming the contract to `F1155`, which is the name used in the white paper.
- Consider renaming `updateState` to `updateIndex`, which more accurately describes what the function does.

`FLiquidator`

- Consider renaming the state variable `IUniswapV2Router02 public swapper`, as it has the same name as the `Swapper` contract but represents something different.

`Controller`

- Consider renaming the contract to `Refinancer` or `RefinancingController` to emphasize that this contract is responsible for refinancing.

`DyDxFlashLoans`

- Consider renaming the `DyDxFlashLoans.sol` filename to avoid inconsistency with the `DyDxFlashloanBase` contract.

F. Index Construction for Interest Calculation

The Fuji Protocol takes out loans from borrowing protocols like Aave and Compound. As these loans accrue interest over time, debt increases as time progresses.

If a user takes out a loan from the Fuji Protocol, she will later have to repay more than the initial value of the loan due to the interest that has accumulated. The exact value she will have to repay is determined by the floating interest rate of the active borrowing protocol during the time between the borrowing and repayment operations.

The interest rate is not fixed but changes based on the supply and demand of the active borrowing protocol's assets. Therefore, the interest rate can be different for each block.

A naive strategy for handling variable interest rates would be to measure the interest rate on each block and adjust each user's debt accordingly. However, this strategy would require one transaction per block, and the gas costs would scale linearly with the number of users. The resulting gas requirements and costs make this solution impractical on Ethereum.

Rather than updating user debt balances on each block, the Fuji Protocol updates them before any operation that depends on up-to-date user debt balances. The interest rate r_t at time t is calculated using the growth between the previous debt balance and the current debt balance owed to the active borrowing protocol: $r_t = b_t / b_{t-1}$. The Fuji internal debt balance d_t of the user is then adjusted by the interest rate: $d_t = d_{t-1} * r_t$.

However, each user's debt must still be updated individually, resulting in gas requirements that scale with the number of users. The active borrowing protocol gives the same interest to all borrowers at any given point in time. As a result, the interest rate r_t is identical for all loans. Therefore, $r_t = (b_t - b_{t-1})/b_{t-1} = (B_t - B_{t-1})/B_{t-1}$, where $b$is an individual user's debt, and $B$is the total debt that the Fuji Protocol owes to the active borrowing protocol.

From this, it follows that instead of updating each user's balance, one can calculate the product of all interest rates r_t in an index that represents the interest rate of a loan that was borrowed at the beginning of the protocol's lifetime: $I_t = 1 * r_1 * r_2 * r_3 * ... * r_t$. One can then multiply a user's initial debt balance d_0 by I_t to obtain the user's current debt balance: $d_t = I_t * d_0$. The index I_t starts at $I_0 = 1$. As a result, a user's initial debt balance is multiplied by the interest rate only when a user’s debt balance is requested, not when the interest rate is updated; this process can be implemented much more efficiently on Ethereum.

Not all users take out their loans at the beginning of the lifetime of the protocol. If a user takes out her loan at timestamp $w! = 0$, then the calculation $d_t = I_t * d_w$ is incorrect, as it gives the user the interest accumulated before the time she took out the loan. To adjust for this issue, a snapshot of the index at the time the loan was taken out, $I_w = 1 * r_1 * r_2 * ... * r_{w-1}$, is remembered, and the user’s debt balance is divided by I_w to divide out the interest rate before the loan is taken out:

$$I_t = 1 * r_1 * r_2 * ... * r_{w-1} * r_w * r_{w+1} * r_{w+2} * ... * r_t$$

$$\Leftrightarrow$$

$$r_w * r_{w+1} * r_{w+2} * ... * r_t = I_t/(1 * r_1 * r_2 * ... * r_{w-1})$$

The following is the resulting formula for calculating any user’s current debt balance:

$$d_t = d_w * I_t/I_w$$

G. Handling Key Material

The safety of key material is important in any system, but particularly so in Ethereum; keys dictate access to money and resources. Theft of keys could mean a complete loss of funds or trust in the market. The current configuration uses an environment variable in production to relay key material to applications that use these keys to interact with on-chain components. However, attackers with local access to the machine may be able to extract these environment variables and steal key material, even without privileged positions. Therefore, we recommend the following:

- Move key material from environment variables to a dedicated secret management system with trusted computing capabilities. The two best options for this are Google Cloud Key Management System (GCKMS) and Hashicorp Vault with hardware security module (HSM) backing.
- Restrict access to GCKMS or Hashicorp Vault to only those applications and administrators that must have access to the credential store.
- Local key material, such as keys used by fund administrators, may be stored in local HSMs, such as YubiHSM2.
- Limit the number of staff members and applications with access to this machine.
- Segment the machine away from all other hosts on the network.
- Ensure strict host logging, patching, and auditing policies are in place for any machine or application that handles said material.
- Determine the business risk of a lost or stolen key, and determine the disaster recovery and business continuity (DR/BC) policies in the event of a stolen or lost key.

H. Fix Log

On December 3, 2021, Trail of Bits reviewed the fixes and mitigations implemented by the Fuji Protocol team for the issues identified in this report. The Fuji Protocol team fixed 13 of the issues reported in the original assessment, partially fixed 4, and acknowledged but did not fix the remaining 4. We reviewed each of the fixes to ensure that the proposed remediation would be effective. The fix commits often contained additional changes not related to the fixes. We did not comprehensively review these changes. For additional information, please refer to the Detailed Fix Log.

ID	Title	Severity	Fix Status
1	Anyone can destroy the FujiVault logic contract if its initialize function was not called during deployment	High	Partially Fixed (`f6858e8`)
2	Providers are implemented with delegatecall	Informational	Risk accepted by the client
3	Lack of contract existence check on delegatecall will result in unexpected behavior	High	Partially fixed (`03a4aa0`)
4	FujiVault.setFactor is unnecessarily complex and does not properly handle invalid input	Informational	Fixed (`9e79d2e`)
5	Preconditions specified in docstrings are not checked by functions	Informational	Fixed (`2efc1b4`, `9e79d2e`)
6	The FujiERC1155.burnBatch function implementation is incorrect	High	Fixed (`900f7d7`)
7	Error in the white paper's equation for the cost of refinancing	Informational	Fixed (`33c8c8b`)
8	Errors in the white paper's equation for index calculation	Medium	Fixed (`33c8c8b`)
9	FujiERC1155.setURI does not adhere to the EIP-1155 specification	Informational	Partially fixed (`3477e6a`)

10	Partial refinancing operations can break the protocol	Medium	Fixed (efa86b0)
11	Native support for ether increases the codebase’s complexity	Informational	Risk accepted by the client
12	Missing events for critical operations	Low	Fixed (3477e6a)
13	Indexes are not updated before all operations that require up-to-date indexes	High	Partially fixed (d17cd77)
14	No protection against missing index updates before operations that depend on up-to-date indexes	Informational	Risk accepted by the client
15	Formula for index calculation is unnecessarily complex	Informational	Fixed (0cc7032)
16	Flasher’s initiateFlashloan function does not revert on invalid flashnum values	Low	Fixed (3cc8b21)
17	Docstrings do not reflect functions’ implementations	Low	Fixed (9e79d2e)
18	Harvester’s getHarvestTransaction function does not revert on invalid _farmProtocolNum and harvestType values	Low	Fixed (794e5d7)
19	Lack of data validation in Controller’s doRefinancing function	Low	Fixed (2efc1b4)
20	Lack of data validation on function parameters	Low	Fixed (293d9aa, 2c96c16, 0d77944)
21	Solidity compiler optimizations can be problematic	Informational	Risk accepted by the client

Detailed Fix Log

TOB-FUJI-001: Anyone can destroy the FujiVault logic contract if its initialize function was not called during deployment
Partially fixed. The Fuji Protocol team modified the `deployVault.js` script to call `initialize` on the vault if it has not been called already. However, the team did not address the root cause of the issue: the dangerous call to `delegatecall`. (`f6858e8`)

TOB-FUJI-002: Providers are implemented with delegatecall
Risk accepted by the client. The Fuji Protocol team provided the following rationale for its acceptance of this risk:

"The team assumes the risk of maintaining the use of delegatecall; however, the rationale comes after some mitigations and an internal analysis with the following statements:

- Debt positions are not transferable as deposit positions are.
- As explained in the report, it is important that the context of the caller at the underlying lending-borrowing protocols is the `FujiVault`. Deposit receipt tokens can easily be transferred, however, debt positions cannot. Aave, has a credit delegation feature that could be used to maintain the desired context, but the remaining providers do not have it. To keep call methods universal for all providers it is preferred that all providers are called similarly and delegatecall facilitates this.
- Mitigations to the risks of delegatecall:
- `_execute` function was changed to private. It now can only be called by only the functions within `VaultBaseUpgradeable.sol`. This means that future upgrades could easily be checked to maintain `_execute` only within the context of `VaultBaseUpgradeable`.
- It was verified that `_execute` can only call addresses defined by two functions; these are set in:
  - `setProviders()` which is restricted to owner
  - `setActiveProvider()` which is restricted to owner and the controller after a refinancing.
- A check was introduced in `executeSwitch` to ensure address passed is a valid provider."

TOB-FUJI-003: Lack of contract existence check on delegatecall will result in unexpected behavior
Partially fixed. The Fuji Protocol team added a contract existence check before the `delegatecall` in `VaultBaseUpgradeable._execute`. Also, assembly is no longer used for the implementation of that `delegatecall`. This makes the call simpler and less error-prone. However, the team is still using OpenZeppelin's `Proxy` contract, which does not check for contract existence before its `delegatecall`. (`03a4aa0`)

TOB-FUJI-004: FujiVault.setFactor is unnecessarily complex and does not properly handle invalid input

Fixed. The Fuji Protocol team modified the `FujiVault.setFactor` function so that it reverts if an invalid type is provided. (`9e79d2e`)

TOB-FUJI-005: Preconditions specified in docstrings are not checked by functions

Fixed. The Fuji Protocol team added the missing checks to `Controller.doRefinancing` and `FujiVault.setFactor`. (`2efc1b4`, `9e79d2e`)

TOB-FUJI-006: The FujiERC1155.burnBatch function implementation is incorrect

Fixed. The Fuji Protocol team replaced `amount` with `amountScaled` in the `burnBatch` and `mintBatch` functions. Additionally, the team extracted the `_mint` and `_burn` functions, which are now reused, leading to less duplicate code and decreasing the chance that similar issues will arise in the burning and minting functions in the future. (`900f7d7`)

TOB-FUJI-007: Error in the white paper's equation for the cost of refinancing

Fixed. The Fuji Protocol team fixed equation 4 in the white paper. Additionally, the team fixed mistakes in the equation beyond those that we reported. However, the text below the equation in the white paper is inconsistent with the equation, since FL_{fee} is now always a percentage. (`33c8c8b`)

TOB-FUJI-008: Errors in the white paper's equation for index calculation

Fixed. The Fuji Protocol team fixed equation 1 in the white paper. (`33c8c8b`)

TOB-FUJI-009: FujiERC1155.setURI does not adhere to the EIP-1155 specification

Partially fixed. `FujiERC1155`'s `setURI` function still does not emit the `URI` event. The decision not to add this event is understandable, as doing so would require looping over all tokens. However, the Fuji Protocol team documented this behavior and modified the function so that it now emits a custom `URIGlobalChanged` event instead. (`3477e6a`)

TOB-FUJI-010: Partial refinancing operations can break the protocol

Fixed. The Fuji Protocol team removed the ability to perform partial refinancing operations from the protocol. (`efa86b0`)

TOB-FUJI-011: Native support for ether increases the codebase's complexity

Risk accepted by the client. The Fuji Protocol team acknowledged the issue and provided the following rationale for its acceptance of this risk:

"Fuji protocol on Ethereum mainnet has to interact with Compound, which cETH market, operates in native ETH. To support ETH as collateral in Compound the asset must be handled in native form. Excluding Compound as a provider is not an option for the Fuji protocol at the moment."

TOB-FUJI-012: Missing events for critical operations
Fixed. The Fuji Protocol team added all missing events to the FujiAdmin, Controller, FujiERC1155, FujiMapping, FujiOracle, and FujiVault contracts. (3477e6a)

TOB-FUJI-013: Indexes are not updated before all operations that require up-to-date indexes
Partially fixed. The Fuji Protocol team added a call to updateF1155Balances in the deposit function but not in the paybackLiq function. The fix commit also contained some refactoring. We did not have time to check the correctness of this refactoring. (d17cd77)

TOB-FUJI-014: No protection against missing index updates before operations that depend on up-to-date indexes
Risk accepted by the client. The Fuji protocol team provided the following rationale for its acceptance of this risk:

"The team acknowledges the recommendation; however, the exploit scenario is limited. The team with foresee no future functions in the design pipeline that will involve index value calls. Item point will be considered for future implementation when there is a architecture design change in the protocol."

TOB-FUJI-015: Formula for index calculation is unnecessarily complex
Fixed. The Fuji Protocol team correctly implemented the simpler formula suggested in this report. (0cc7032)

TOB-FUJI-016: Flasher's initiateFlashloan function does not revert on invalid flashnum values
Fixed. The initiateFlashloan function now reverts on invalid _flashnum values. (3cc8b21)

TOB-FUJI-017: Docstrings do not reflect functions' implementations
Fixed. The Fuji Protocol team updated the docstrings so that they reflect the implementations. (9e79d2e)

TOB-FUJI-018: Harvester's getHarvestTransaction function does not revert on invalid _farmProtocolNum and harvestType values
Fixed. The VaultHarvester contract's getHarvestTransaction function now reverts on invalid _farmProtocolNum and harvestType values. (794e5d7)

TOB-FUJI-019: Lack of data validation in Controller's doRefinancing function
Fixed. The Controller contract's doRefinancing function now reverts if the new provider equals the active provider. (2efc1b4)

TOB-FUJI-020: Lack of data validation on function parameters

Fixed. The Fuji Protocol team added missing zero address checks to the following functions: `Claimable.transferOwnership`, `ClaimableUpgradeable.transferOwnership`, `Controller.setFujiAdmin`, `FujiVault.setFujiAdmin`, `FujiVault.setProviders`, and `FujiVault.setOracle`. As of commit `0d77944`, the `FujiAdmin` contract is not missing any checks. The team also added further zero checks. (`293d9aa`, `2c96c16`, `0d77944`)

TOB-FUJI-021: Solidity compiler optimizations can be problematic

Risk accepted by the client.