Fuji Protocol

Date	March 2022
Auditors	Dominik Muhs, Martin Ortner

1 Executive Summary

This report presents the results of our engagement with **FujiDAO** to review the **Fuji Protocol**, a borrowing aggregator protocol.

The review was conducted over two weeks, from **February 21, 2022** to **March 04, 2022**, and 2x2 person weeks were spent.

Starting with an investigation into the overall architecture of the system, the assessment team dissected the system into its main components. In a kick-off call on Tuesday, February 22, 2022, it was mutually agreed that the `fantom` contracts are the main priority. However, anything found with the `mainnet` (Ethereum) contracts will also be reported as we come by findings.

The team continued reviewing the contracts individually based on the components' perceived risk profile, followed by reviewing the entire system that started in the second week. The client provided a Whitepaper and a high-level user-facing documentation.

Closing the first week of the review, we reached out to the client with a request to help assess the severity of a potential security issue with the flash loan contract. Following this, the client chose to upgrade one of their contracts implementing a hotfix to mitigate the potential of unsolicited flash loans executing privileged functionality in the system.

It should be noted that the `fantom` and `mainnet/Ethereum` contracts are very similar. The findings listed in this report may affect both code-bases.

Due to the amount and classes of findings reported, we highly recommend addressing the results reported followed by a thorough review of the next iteration before going live.

2 Scope

Our review focused on the commit hash #f8436a81437914d43297761b4011d60b21f17216. The list of files in scope can be found in the Appendix.

Initially, all contracts in ./contracts were in scope. However, due to the time-boxed nature of this review, it was agreed that the main priority should be the contracts in the ./contracts/fantom folder while still reporting issues with mainnet contracts as we come across them.

2.1 Objectives

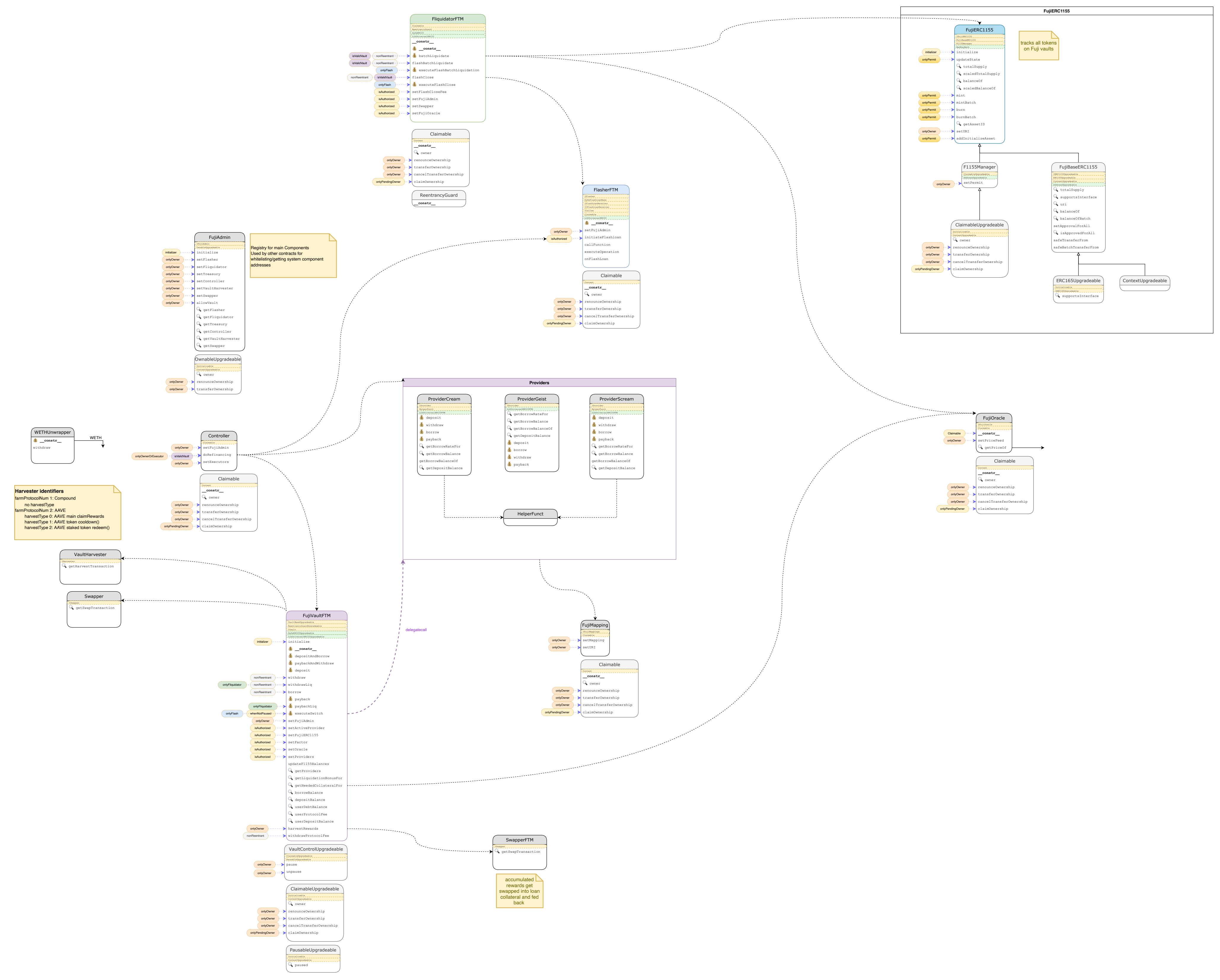
Together with the **FujiDAO** team, we identified the following priorities for our review:

1. Ensure that the system is implemented consistently with the intended functionality and without unintended edge cases.
2. Identify known vulnerabilities particular to smart contract systems, as outlined in our Smart Contract Best Practices, and the Smart Contract Weakness Classification Registry.

3 System Overview

This section describes the top-level/deployable contracts, their inheritance structure and interfaces, actors, permissions, and essential contract interactions of the system under review.

Contracts are depicted as boxes. Publicly reachable interface methods are outlined as rows in each box. The 🔍 icon indicates that a method is declared non-state-changing (view/pure), while other methods may change state. A yellow dashed row at the top of the contract shows inherited contracts. A green dashed row at the top of the contract indicates referenced libraries. Access control modifiers are connected as yellow "gatekeeper"-bubbles in front of methods. The owner of various components is depicted as an actor symbol.



4 Findings

Each issue has an assigned severity:

- Minor issues are subjective in nature. They are typically suggestions around best practices or readability. Code maintainers should use their own judgment as to whether to address such issues.
- Medium issues are objective in nature but are not security vulnerabilities. These should be addressed unless there is a clear reason not to.
- Major issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- Critical issues are directly exploitable security vulnerabilities that need to be fixed.

4.1 FlasherFTM - Unsolicited invocation of the callback (CREAM auth bypass) Critical

Description

TL;DR: Anyone can call `ICTokenFlashloan(crToken).flashLoan(address(FlasherFTM), address(FlasherFTM), info.amount, params)` directly and pass validation checks in `onFlashLoan()`. This call forces it to accept unsolicited flash loans and execute the actions provided under the attacker's `FlashLoan.Info`.

`receiver.onFlashLoan(initiator, token, amount, ...)` is called when receiving a flash loan. According to EIP-3156, the `initiator` is `msg.sender` so that one can use it to check if the call to `receiver.onFlashLoan()` was unsolicited or not.

Third-party Flash Loan provider contracts are often upgradeable.

For example, the Geist lending contract configured with this system is upgradeable. Upgradeable contracts bear the risk that one cannot assume that the contract is always running the same code. In the worst case, for example, a malicious proxy admin (leaked keys, insider, …) could upgrade the contract and perform unsolicited calls with arbitrary data to Flash Loan consumers in an attempt to exploit them. It, therefore, is highly recommended to verify that flash loan callbacks in the system can only be called if the contract was calling out to the provider to provide a Flash Loan and that the conditions of the flash loan (returned data, amount) are correct.

Not all Flash Loan providers implement EIP-3156 correctly.

Cream Finance, for example, allows users to set **an arbitrary initiator** when requesting a flash loan. This **deviates from EIP-3156** and was reported to the Cream development team as a security issue. Hence, anyone can spoof that `initiator` and potentially bypass authentication checks in the consumers' `receiver.onFlashLoan()`. Depending on the third-party application consuming the flash loan is doing with the funds, the impact might range from medium to critical with funds at risk. For example, projects might assume that the flash loan always originates from their trusted components, e.g., because they use them to refinance switching funds between pools or protocols.

Examples

- The `FlasherFTM` contract assumes that flash loans for the Flasher can only be initiated by authorized callers (`isAuthorized`) - for a reason - because it is vital that the `FlashLoan.Info calldata info` parameter only contains trusted data:

code/contracts/fantom/flashloans/FlasherFTM.sol:L66-L79

```
/**
 * @dev Routing Function for Flashloan Provider
 * @param info: struct information for flashLoan
 * @param _flashnum: integer identifier of flashloan provider
 */
function initiateFlashloan(FlashLoan.Info calldata info, uint8 _flashnum) external isAuthorized override {
  if (_flashnum == 0) {
    _initiateGeistFlashLoan(info);
  } else if (_flashnum == 2) {
    _initiateCreamFlashLoan(info);
  } else {
    revert(Errors.VL_INVALID_FLASH_NUMBER);
  }
}
```

code/contracts/fantom/flashloans/FlasherFTM.sol:L46-L55

```
modifier isAuthorized() {
  require(
    msg.sender == _fujiAdmin.getController() ||
      msg.sender == _fujiAdmin.getFliquidator() ||
      msg.sender == owner(),
    Errors.VL_NOT_AUTHORIZED
  );
  _;
}
```

- The Cream Flash Loan initiation code requests the flash loan via `ICTokenFlashloan(crToken).flashLoan(receiver=address(this), initiator=address(this), ...)`:

code/contracts/fantom/flashloans/FlasherFTM.sol:L144-L158

```
/**
 * @dev Initiates an CreamFinance flashloan.
 * @param info: data to be passed between functions executing flashloan logic
 */
function _initiateCreamFlashLoan(FlashLoan.Info calldata info) internal {
  address crToken = info.asset == _FTM
    ? 0xd528697008aC67A21818751A5e3c58C8daE54696
    : _crMappings.addressMapping(info.asset);

  // Prepara data for flashloan execution
  bytes memory params = abi.encode(info);

  // Initialize Instance of Cream crLendingContract
  ICTokenFlashloan(crToken).flashLoan(address(this), address(this), info.amount, params);
}
```

Note: The Cream implementation does not send `sender=msg.sender` to the `onFlashLoan()` callback - like any other flash loan provider does and EIP-3156 suggests - but uses the value that was passed in as `initiator` when requesting the callback. This detail completely undermines the authentication checks implemented in `onFlashLoan` as the `sender` value cannot be trusted.

contracts/CCollateralCapErc20.sol:L187

```
address initiator,
```

code/contracts/fantom/flashloans/FlasherFTM.sol:L162-L175

```
 */
function onFlashLoan(
  address sender,
  address underlying,
  uint256 amount,
  uint256 fee,
  bytes calldata params
) external override returns (bytes32) {
  // Check Msg. Sender is crToken Lending Contract
  // from IronBank because ETH on Cream cannot perform a flashloan
  address crToken = underlying == _WFTM
    ? 0xd528697008aC67A21818751A5e3c58C8daE54696
    : _crMappings.addressMapping(underlying);
  require(msg.sender == crToken && address(this) == sender, Errors.VL_NOT_AUTHORIZED);
```

Recommendation

Cream Finance

We've reached out to the Cream developer team, who have confirmed the issue. They are planning to implement countermeasures. Our recommendation can be summarized as follows:

> Implement the EIP-3156 compliant version of flashLoan() with initiator hardcoded to `msg.sender`.

FujiDAO (and other flash loan consumers)

We recommend not assuming that `FlashLoan.Info` contains trusted or even validated data when a third-party flash loan provider provides it! Developers should ensure that the data received was provided when the flash loan was requested.

The contract should reject unsolicited flash loans. In the scenario where a flash loan provider is exploited, the risk of an exploited trust relationship is less likely to spread to the rest of the system.

The Cream `initiator` provided to the `onFlashLoan()` callback cannot be trusted until the Cream developers fix this issue. The initiator can easily be spoofed to perform unsolicited flash loans. We, therefore, suggest:

1. Validate that the `initiator` value is the `flashLoan()` caller. This conforms to the standard and is hopefully how the Cream team is fixing this, and
2. Ensure the implementation tracks its own calls to `flashLoan()` in a state-variable semaphore, i.e. store the flash loan data/hash in a temporary state-variable that is only set just before calling `flashLoan()` until being called back in `onFlashLoan()`. The received data can then be verified against the stored artifact. This is a safe way of authenticating and verifying callbacks.

Values received from untrusted third parties should always be validated with the utmost scrutiny.

Smart contract upgrades are risky, so we recommend implementing the means to pause certain flash loan providers.

Ensure that flash loan handler functions should never re-enter the system. This provides additional security guarantees in case a flash loan provider gets breached.

Note: The Fuji development team implemented a hotfix to prevent unsolicited calls from Cream by storing the `hash(FlashLoan.info)` in a state variable just before requesting the flash loan. Inside the `onFlashLoan` callback, this state is validated and cleared accordingly.

An improvement to this hotfix would be, to check `_paramsHash` before any external calls are made and clear it right after validation at the beginning of the function. Additionally, `hash==0x0` should be explicitly disallowed. By doing so, the check also serves as a reentrancy guard and helps further reduce the risk of a potentially malicious flash loan re-entering the function.

4.2 Lack of reentrancy protection in token interactions Major

Description

Token operations may potentially re-enter the system. For example, `univTransfer` may perform a low-level `to.call{value}()` and, depending on the token's specification (e.g. `ERC-20` extension or `ERC-20` compliant `ERC-777`), `token` may implement callbacks when being called as `token.safeTransfer(to, amount)` (or `token.transfer*()`).

Therefore, it is crucial to strictly adhere to the checks-effects pattern and safeguard affected methods using a mutex.

Examples

code/contracts/fantom/libraries/LibUniversalERC20FTM.sol:L26-L40

```
function univTransfer(
  IERC20 token,
  address payable to,
  uint256 amount
) internal {
  if (amount > 0) {
    if (isFTM(token)) {
      (bool sent, ) = to.call{ value: amount }("");
      require(sent, "Failed to send Ether");
    } else {
      token.safeTransfer(to, amount);
    }
  }
}
```

- `withdraw` is `nonReentrant` while `paybackAndWithdraw` is not, which appears to be inconsistent

code/contracts/fantom/FujiVaultFTM.sol:L172-L182

```
/**
 * @dev Paybacks the underlying asset and withdraws collateral in a single function call from activeProvider
 * @param _paybackAmount: amount of underlying asset to be payback, pass -1 to pay full amount
 * @param _collateralAmount: amount of collateral to be withdrawn, pass -1 to withdraw maximum amount
 */
function paybackAndWithdraw(int256 _paybackAmount, int256 _collateralAmount) external payable {
  updateF1155Balances();
  _internalPayback(_paybackAmount);
  _internalWithdraw(_collateralAmount);
}
```

code/contracts/fantom/FujiVaultFTM.sol:L232-L241

```
/**
 * @dev Paybacks Vault's type underlying to activeProvider - called by users
 * @param _repayAmount: token amount of underlying to repay, or
 * pass any 'negative number' to repay full ammount
 * Emits a {Repay} event.
 */
function payback(int256 _repayAmount) public payable override {
  updateF1155Balances();
  _internalPayback(_repayAmount);
}
```

- `depositAndBorrow` is not `nonReentrant` while `borrow()` is which appears to be inconsistent

code/contracts/fantom/FujiVaultFTM.sol:L161-L171

```
/**
 * @dev Deposits collateral and borrows underlying in a single function call from activeProvider
 * @param _collateralAmount: amount to be deposited
 * @param _borrowAmount: amount to be borrowed
 */
function depositAndBorrow(uint256 _collateralAmount, uint256 _borrowAmount) external payable {
  updateF1155Balances();
  _internalDeposit(_collateralAmount);
  _internalBorrow(_borrowAmount);
}
```

code/contracts/fantom/FujiVaultFTM.sol:L222-L230

```
/**
 * @dev Borrows Vault's type underlying amount from activeProvider
 * @param _borrowAmount: token amount of underlying to borrow
 * Emits a {Borrow} event.
 */
function borrow(uint256 _borrowAmount) public override nonReentrant {
  updateF1155Balances();
  _internalBorrow(_borrowAmount);
}
```

Here's an example call stack for `depositAndBorrow` that outlines how a reentrant `ERC20` token (e.g. `ERC777`) may call back into `depositAndBorrow` again, `updateBalances` twice in the beginning before tokens are even transferred and then continues to call `internalDeposit`, `internalBorrow`, `internalBorrow` without an update before the 2nd borrow. Note that both `internalDeposit` and `internalBorrow` read indexes that may now be outdated.

```
depositAndBorrow
    updateBalances
    internalDeposit ->
                        ERC777(collateralAsset).safeTransferFrom()  ---> calls back!
                        ---callback:beforeTokenTransfer---->
                                !! depositAndBorrow
                                        updateBalances
                                        internalDeposit
                                            --> ERC777.safeTransferFrom()
                                            <--
                                            _deposit
                                            mint
                                        internalBorrow
                                            mint
                                            _borrow
                                            ERC777(borrowAsset).univTransfer(msg.sender) --> might call back

                    <-------------------------------
                    _deposit
                    mint
    internalBorrow
        mint
        _borrow
        --> ERC777(borrowAsset).univTransfer(msg.sender) --> might call back
        <--
```

Recommendation

Consider decorating methods that may call back to untrusted sources (i.e., native token transfers, callback token operations) as `nonReentrant` and strictly follow the checks-effects pattern for all contracts in the code-base.

4.3 Lack of segregation of duties, excessive owner permissions, misleading authentication modifiers Major

Descriptio

In the `FujiERC1155` contract, the `onlyPermit` modifier should not include `owner`.

The `FujiERC1155` is `claimable` (`ownable`) via `F1155Manager`. The `onlyPermit` modifier includes contracts explicitly permitted to perform actions, and the `owner`, in a lot of cases, has separate duties. Note that the `owner` can add new contracts to the `onlyPermit` whitelist.

code/contracts/abstracts/fujiERC1155/F1155Manager.sol:L34-L37

```
modifier onlyPermit() {
  require(addrPermit[_msgSender()] || msg.sender == owner(), Errors.VL_NOT_AUTHORIZED);
  _;
}
```

However, the `owner` can also wholly mess up accounting as they are permitted to call `updateState()`, which should only be callable by vaults:

code/contracts/FujiERC1155.sol:L53-L59

```
function updateState(uint256 _assetID, uint256 newBalance) external override onlyPermit {
  uint256 total = totalSupply(_assetID);
  if (newBalance > 0 && total > 0 && newBalance > total) {
    uint256 newIndex = (indexes[_assetID] * newBalance) / total;
    indexes[_assetID] = uint128(newIndex);
  }
}
```

The same is true for `FujiERC1155.{mint|mintBatch|burn|burnBatch|addInitializeAsset}` unless there is a reason to allow `owner` to freely burn/mint/initialize tokens and updateState for borrowed assets to arbitrary values.

- `FujiVault` - `owner` is part of `isAuthorized` and can change the system out-of-band. `controller` does not implement means to call functions it has permissions to.

Multiple methods in FujiVault are decorated with the access control `isAuthorized` that grants the `owner` and the currently configured `controller` access. The `controller`, however, does not implement any means to call some of the methods on the Vault.

Furthermore, the owner is part of `isAuthorized`, too, and can switch out the debt-management token while one is already configured without any migration. This is likely to create an inconsistent state with the Vault, and no one will be able to withdraw their now non-existent token.

code/contracts/fantom/FujiVaultFTM.sol:L65-L74

```
/**
 * @dev Throws if caller is not the 'owner' or the '_controller' address stored in {FujiAdmin}
 */
modifier isAuthorized() {
  require(
    msg.sender == owner() || msg.sender == _fujiAdmin.getController(),
    Errors.VL_NOT_AUTHORIZED
  );
  _;
}
```

The `owner` can call methods “out of band,” bypassing steps the contract system would enforce otherwise, e.g. `controller` calling `setActiveProvider`.

It is assumed that `setOracle`, `setFactor` should probably be `onlyOwner` instead.

code/contracts/fantom/FujiVaultFTM.sol:L354-L367

```
function setFujiERC1155(address _fujiERC1155) external isAuthorized {
  require(_fujiERC1155 != address(0), Errors.VL_ZERO_ADDR);
  fujiERC1155 = _fujiERC1155;

  vAssets.collateralID = IFujiERC1155(_fujiERC1155).addInitializeAsset(
    IFujiERC1155.AssetType.collateralToken,
    address(this)
  );
  vAssets.borrowID = IFujiERC1155(_fujiERC1155).addInitializeAsset(
    IFujiERC1155.AssetType.debtToken,
    address(this)
  );
  emit F1155Changed(_fujiERC1155);
}
```

Note ensure that `setProviders` can only ever be set by a trusted entity or multi-sig as the `Vault` delegatecalls the provider logic (via `VaultControlUpgradeable`) and, hence, the provider has total control over the `Vault` storage!

- `FliquidatorFTM` - Unnecessary and confusing modifier `FliquidatorFTM.isAuthorized`

The contract is already `Claimable`; therefore, use the already existing modifier `Claimable.onlyOwner` instead.

code/contracts/fantom/FliquidatorFTM.sol:L86-L91

```
*/
modifier isAuthorized() {
  require(msg.sender == owner(), Errors.VL_NOT_AUTHORIZED);
  _;
}
```

code/contracts/abstracts/claimable/Claimable.sol:L48-L51

```
modifier onlyOwner() {
  require(_msgSender() == owner(), "Ownable: caller is not the owner");
  _;
}
```

Use `Claimable.onlyOwner` instead.

- `FlasherFTM` - `owner` should not be able to call `initiateFlashloan` directly; misleading comment.

code/contracts/fantom/flashloans/FlasherFTM.sol:L42-L54

```

/**
* @dev Throws if caller is not 'owner'.
*/
modifier isAuthorized() {
  require(
    msg.sender == _fujiAdmin.getController() ||
      msg.sender == _fujiAdmin.getFliquidator() ||
      msg.sender == owner(),
    Errors.VL_NOT_AUTHORIZED
  );
  _;
}
```

- `FujiERC1155` - All vaults have equal permission to `mint/burn/initializeAssets` for every vault

All vaults need to be in the `onlyPermit` ACL whitelist. No additional checks enforce that the calling vault can only modify its token balances. Furthermore, `FujiVaultFTM` is upgradeable; thus, the contract logic may be altered to allow the vault to modify any other token id's balance. To reduce this risk and the potential of an exploited contract affecting other token balances in the system, it is suggested to change the coarse `onlyPermit` ACL to one that checks that the calling vault can only manage their token IDs.

Recommendation

Reconsider the authentication concept and make it more transparent. Segregate duties and clearly define roles and capabilities. Avoid having overly powerful actors and reduce their capabilities to the bare minimum needed to segregate risk. If an actor is part of an ACL in a third-party contract, they also should have the means to call that method in a controlled way or else remove them from the ACL. To avoid conveying a false sense of trust towards certain actors within the smart contract system, it is suggested to use the centralized `onlyOwner` decorator for methods only the owner can call. This more accurately depicts "who can do what" in the system and makes it easier to trust the project team managing it.

Avoid excessively powerful `owners` that can change/mint/burn anything in the system as this is a risk for the general consistency.

Remove `owner` from methods/modifiers they don't need to be part of/have access to.

Ensure `owner` is a time-locked multi-sig or governance contract. Rename authentication modifiers to describe better what callers they allow.

4.4 Unchecked Return Values - ICErc20 `repayBorrow` Major

Description

`ICErc20.repayBorrow` returns a non-zero uint on error. Multiple providers do not check for this error condition and might return `success` even though `repayBorrow` failed, returning an error code.

This can potentially allow a malicious user to call `paybackAndWithdraw()` while not repaying by causing an error in the sub-call to `Compound.repayBorrow()`, which ends up being silently ignored. Due to the missing success condition check, execution continues normally with `_internalWithdraw()`.

Also, see issue 4.5.

code/contracts/interfaces/compound/ICErc20.sol:L11-L12

```
function repayBorrow(uint256 repayAmount) external returns (uint256);
```

The method may return an error due to multiple reasons:

contracts/CToken.sol:L808-L816

```
function repayBorrowInternal(uint repayAmount) internal nonReentrant returns (uint, uint) {
    uint error = accrueInterest();
    if (error != uint(Error.NO_ERROR)) {
        // accrueInterest emits logs on errors, but we still want to log the fact that an attempted borrow failed
        return (fail(Error(error), FailureInfo.REPAY_BORROW_ACCRUE_INTEREST_FAILED), 0);
    }
    // repayBorrowFresh emits repay-borrow-specific logs on errors, so we don't need to
    return repayBorrowFresh(msg.sender, msg.sender, repayAmount);
}
```

contracts/CToken.sol:L855-L873

```
if (allowed != 0) {
    return (failOpaque(Error.COMPTROLLER_REJECTION, FailureInfo.REPAY_BORROW_COMPTROLLER_REJECTION, allowed), 0);
}

/* Verify market's block number equals current block number */
if (accrualBlockNumber != getBlockNumber()) {
    return (fail(Error.MARKET_NOT_FRESH, FailureInfo.REPAY_BORROW_FRESHNESS_CHECK), 0);
}

RepayBorrowLocalVars memory vars;

/* We remember the original borrowerIndex for verification purposes */
vars.borrowerIndex = accountBorrows[borrower].interestIndex;

/* We fetch the amount the borrower owes, with accumulated interest */
(vars.mathErr, vars.accountBorrows) = borrowBalanceStoredInternal(borrower);
if (vars.mathErr != MathError.NO_ERROR) {
    return (failOpaque(Error.MATH_ERROR, FailureInfo.REPAY_BORROW_ACCUMULATED_BALANCE_CALCULATION_FAILED, uint(vars.mathErr)), 0
}
```

Examples

Multiple providers, here are some examples:

code/contracts/fantom/providers/ProviderCream.sol:L168-L173

```
  // Check there is enough balance to pay
  require(erc20token.balanceOf(address(this)) >= _amount, "Not-enough-token");
  erc20token.univApprove(address(cyTokenAddr), _amount);
  cyToken.repayBorrow(_amount);
}
```

code/contracts/fantom/providers/ProviderScream.sol:L170-L172

```
require(erc20token.balanceOf(address(this)) >= _amount, "Not-enough-token");
erc20token.univApprove(address(cyTokenAddr), _amount);
cyToken.repayBorrow(_amount);
```

code/contracts/mainnet/providers/ProviderCompound.sol:L139-L155

```
if (_isETH(_asset)) {
  // Create a reference to the corresponding cToken contract
  ICEth cToken = ICEth(cTokenAddr);

  cToken.repayBorrow{ value: msg.value }();
} else {
  // Create reference to the ERC20 contract
  IERC20 erc20token = IERC20(_asset);

  // Create a reference to the corresponding cToken contract
  ICErc20 cToken = ICErc20(cTokenAddr);

  // Check there is enough balance to pay
  require(erc20token.balanceOf(address(this)) >= _amount, "Not-enough-token");
  erc20token.univApprove(address(cTokenAddr), _amount);
  cToken.repayBorrow(_amount);
}
```

Recommendation

Check for `cyToken.repayBorrow(_amount) != 0` or `Error.NO_ERROR`.

4.5 Unchecked Return Values - IComptroller `exitMarket`, `enterMarket` Major

Description

`IComptroller.exitMarket()`, `IComptroller.enterMarkets()` may return a non-zero uint on error but none of the Providers check for this error condition. Together with issue 4.10, this might suggest that unchecked return values may be a systemic problem.

Here's the upstream implementation:

contracts/Comptroller.sol:L179-L187

```
if (amountOwed != 0) {
    return fail(Error.NONZERO_BORROW_BALANCE, FailureInfo.EXIT_MARKET_BALANCE_OWED);
}

/* Fail if the sender is not permitted to redeem all of their tokens */
uint allowed = redeemAllowedInternal(cTokenAddress, msg.sender, tokensHeld);
if (allowed != 0) {
    return failOpaque(Error.REJECTION, FailureInfo.EXIT_MARKET_REJECTION, allowed);
}
```

```
  /**
     * @notice Removes asset from sender's account liquidity calculation
     * @dev Sender must not have an outstanding borrow balance in the asset,
     *  or be providing necessary collateral for an outstanding borrow.
     * @param cTokenAddress The address of the asset to be removed
     * @return Whether or not the account successfully exited the market
     */
    function exitMarket(address cTokenAddress) external returns (uint) {
        CToken cToken = CToken(cTokenAddress);
        /* Get sender tokensHeld and amountOwed underlying from the cToken */
        (uint oErr, uint tokensHeld, uint amountOwed, ) = cToken.getAccountSnapshot(msg.sender);
        require(oErr == 0, "exitMarket: getAccountSnapshot failed"); // semi-opaque error code

        /* Fail if the sender has a borrow balance */
        if (amountOwed != 0) {
            return fail(Error.NONZERO_BORROW_BALANCE, FailureInfo.EXIT_MARKET_BALANCE_OWED);
        }

        /* Fail if the sender is not permitted to redeem all of their tokens */
        uint allowed = redeemAllowedInternal(cTokenAddress, msg.sender, tokensHeld);
        if (allowed != 0) {
            return failOpaque(Error.REJECTION, FailureInfo.EXIT_MARKET_REJECTION, allowed);
        }
```

Examples

- Unchecked return value `exitMarket`

All Providers exhibit the same issue, probably due to code reuse. (also see https://github.com/ConsenSysDiligence/fuji-protocol-audit-2022-02/issues/19). Some examples:

code/contracts/fantom/providers/ProviderCream.sol:L52-L57

```
function _exitCollatMarket(address _cyTokenAddress) internal {
  // Create a reference to the corresponding network Comptroller
  IComptroller comptroller = IComptroller(_getComptrollerAddress());

  comptroller.exitMarket(_cyTokenAddress);
}
```

code/contracts/fantom/providers/ProviderScream.sol:L52-L57

```
function _exitCollatMarket(address _cyTokenAddress) internal {
  // Create a reference to the corresponding network Comptroller
  IComptroller comptroller = IComptroller(_getComptrollerAddress());

  comptroller.exitMarket(_cyTokenAddress);
}
```

code/contracts/mainnet/providers/ProviderCompound.sol:L46-L51

```
function _exitCollatMarket(address _cTokenAddress) internal {
  // Create a reference to the corresponding network Comptroller
  IComptroller comptroller = IComptroller(_getComptrollerAddress());

  comptroller.exitMarket(_cTokenAddress);
}
```

code/contracts/mainnet/providers/ProviderIronBank.sol:L52-L57

```
function _exitCollatMarket(address _cyTokenAddress) internal {
  // Create a reference to the corresponding network Comptroller
  IComptroller comptroller = IComptroller(_getComptrollerAddress());

  comptroller.exitMarket(_cyTokenAddress);
}
```

- Unchecked return value `enterMarkets` (Note that `IComptroller` returns `NO_ERROR` when already joined to `enterMarkets`.

All Providers exhibit the same issue, probably due to code reuse. (also see https://github.com/ConsenSysDiligence/fuji-protocol-audit-2022-02/issues/19). For example:

code/contracts/fantom/providers/ProviderCream.sol:L39-L46

```
function _enterCollatMarket(address _cyTokenAddress) internal {
  // Create a reference to the corresponding network Comptroller
  IComptroller comptroller = IComptroller(_getComptrollerAddress());

  address[] memory cyTokenMarkets = new address[](1);
  cyTokenMarkets[0] = _cyTokenAddress;
  comptroller.enterMarkets(cyTokenMarkets);
}
```

Recommendation

Require that return value is `ERROR.NO_ERROR` or `0`.

4.6 Fliquidator - excess funds of native tokens are not returned Medium

Description

`FliquidatorFTM.batchLiquidate` accepts the `FTM` native token and checks if at least an amount of `debtTotal` was provided with the call. The function continues using the `debtTotal` value. If a caller provides `msg.value > debtTotal`, excess funds are not returned and remain in the contract. `FliquidatorFTM` is not upgradeable, and there is no way to recover the surplus funds.

Examples

code/contracts/fantom/FliquidatorFTM.sol:L148-L150

```
if (vAssets.borrowAsset == FTM) {
  require(msg.value >= debtTotal, Errors.VL_AMOUNT_ERROR);
} else {
```

Recommendation

Consider returning excess funds. Consider making `_constructParams` public to allow the caller to pre-calculate the `debtTotal` that needs to be provided with the call.

Consider removing support for native token `FTM` entirely to reduce the overall code complexity. The wrapped equivalent can be used instead.

4.7 Unsafe arithmetic casts Medium

Description

The reason for using signed integers in some situations appears to be to use negative values as an indicator to withdraw everything. Using a whole bit of uint256 for this is quite a lot when using `type(uint256).max` would equal or better serve as a flag to withdraw everything.

Furthermore, even though the code uses `solidity 0.8.x`, which safeguards arithmetic operations against under/overflows, arithmetic typecast is not protected.

Also, see issue 4.9 for a related issue.

```
⇒  solidity-shell

🚀 Entering interactive Solidity ^0.8.11 shell. '.help' and '.exit' are your friends.
 »  ℹ️  ganache-mgr: starting temp. ganache instance ...
 »  uint(int(-100))
115792089237316195423570985008687907853269984665640564039457584007913129639836
 »  int256(uint(2**256-100))
-100
```

Examples

code/contracts/fantom/FliquidatorFTM.sol:L167-L178

```
// Compute how much collateral needs to be swapt
uint256 collateralInPlay = _getCollateralInPlay(
  vAssets.collateralAsset,
  vAssets.borrowAsset,
  debtTotal + bonus
);

// Burn f1155
_burnMulti(addrs, borrowBals, vAssets, _vault, f1155);

// Withdraw collateral
IVault(_vault).withdrawLiq(int256(collateralInPlay));
```

code/contracts/fantom/FliquidatorFTM.sol:L264-L276

```
// Compute how much collateral needs to be swapt for all liquidated users
uint256 collateralInPlay = _getCollateralInPlay(
  vAssets.collateralAsset,
  vAssets.borrowAsset,
  _amount + _flashloanFee + bonus
);

// Burn f1155
_burnMulti(_addrs, _borrowBals, vAssets, _vault, f1155);

// Withdraw collateral
IVault(_vault).withdrawLiq(int256(collateralInPlay));
```

code/contracts/fantom/FliquidatorFTM.sol:L334-L334

```
uint256 amount = _amount < 0 ? debtTotal : uint256(_amount);
```

code/contracts/fantom/FujiVaultFTM.sol:L213-L220

```
function withdrawLiq(int256 _withdrawAmount) external override nonReentrant onlyFliquidator {
  // Logic used when called by Fliquidator
  _withdraw(uint256(_withdrawAmount), address(activeProvider));
  IERC20Upgradeable(vAssets.collateralAsset).univTransfer(
    payable(msg.sender),
    uint256(_withdrawAmount)
  );
}
```

- pot. unsafe truncation (unlikely)

code/contracts/FujiERC1155.sol:L53-L59

```
function updateState(uint256 _assetID, uint256 newBalance) external override onlyPermit {
  uint256 total = totalSupply(_assetID);
  if (newBalance > 0 && total > 0 && newBalance > total) {
    uint256 newIndex = (indexes[_assetID] * newBalance) / total;
    indexes[_assetID] = uint128(newIndex);
  }
}
```

Recommendation

If negative values are only used as a flag to indicate that all funds should be used for an operation, use `type(uint256).max` instead. It is wasting less value-space for a simple flag than using the uint256 high-bit range. Avoid typecast where possible. Use `SafeCast` instead or verify that the casts are safe because the values they operate on cannot under- or overflow. Add inline code comments if that's the case.

4.8 Missing input validation on flash close fee factors Medium

Description

The `FliquidatorFTM` contract allows authorized parties to set the flash close fee factor. The factor is provided as two integers denoting numerator and denominator. Due to a lack of boundary checks, it is possible to set unrealistically high factors, which go well above 1. This can have unexpected effects on internal accounting and the impact of flashloan balances.

Examples

code/contracts/fantom/FliquidatorFTM.sol:L657-L659

```
function setFlashCloseFee(uint64 _newFactorA, uint64 _newFactorB) external isAuthorized {
  flashCloseF.a = _newFactorA;
  flashCloseF.b = _newFactorB;
```

Recommendation

Add a requirement making sure that `flashCloseF.a <= flashCloseF.b`.

4.9 Separation of concerns and consistency in vaults Medium

Description

The `FujiVaultFTM` contract contains multiple balance-changing functions. Most notably, `withdraw` is passed an `int256` denoted amount parameter. Negative values of this parameter are given to the `_internalWithdraw` function, where they trigger the withdrawal of all collateral. This approach can result in accounting mistakes in the future as beyond a certain point in the vault's accounting; amounts are expected to be only positive. Furthermore, the concerns of withdrawing and entirely withdrawing are not separated.

The above issue applies analogously to the `payback` function and its dependency on `_internalPayback`.

For consistency, `withdrawLiq` also takes an `int256` amount parameter. This function is only accessible to the `Fliquidator` contract and withdraws collateral from the active provider. However, all occurrences of the `_withdrawAmount` parameter are cast to `uint256`.

Examples

The `withdraw` entry point:

code/contracts/fantom/FujiVaultFTM.sol:L201-L204

```
function withdraw(int256 _withdrawAmount) public override nonReentrant {
  updateF1155Balances();
  _internalWithdraw(_withdrawAmount);
}
```

`_internalWithdraw`'s negative amount check:

code/contracts/fantom/FujiVaultFTM.sol:L654-L657

```
uint256 amountToWithdraw = _withdrawAmount < 0
  ? providedCollateral - neededCollateral
  : uint256(_withdrawAmount);
```

The `withdrawLiq` entry point for the `Fliquidator`:

code/contracts/fantom/FujiVaultFTM.sol:L213-L220

```
function withdrawLiq(int256 _withdrawAmount) external override nonReentrant onlyFliquidator {
  // Logic used when called by Fliquidator
  _withdraw(uint256(_withdrawAmount), address(activeProvider));
  IERC20Upgradeable(vAssets.collateralAsset).univTransfer(
    payable(msg.sender),
    uint256(_withdrawAmount)
  );
}
```

Recommendation

We recommend splitting the `withdraw(int256)` function into two: `withdraw(uint256)` and `withdrawAll()`. These will provide the same functionality while rendering the updated code of `_internalWithdraw` easier to read, maintain, and harder to manipulate. The recommendation applies to `payback` and `_internalPayback`.

Similarly, `withdrawLiq`'s parameter should be a `uint256` to prevent unnecessary casts.

4.10 Aave/Geist Interface declaration mismatch and unchecked return values Medium

Description

The two lending providers, Geist & Aave, do not seem to be directly affiliated even though one is a fork of the other. However, the interfaces may likely diverge in the future. Using the same interface declaration for both protocols might become problematic with future upgrades to either protocol. The interface declaration does not seem to come from the original upstream project. The interface `IAaveLendingPool` does not declare any return values while some of the functions called in Geist or Aave return them.

Note: that we have not verified all interfaces for correctness. However, we urge the client to only use official interface declarations from the upstream projects and verify that all other interfaces match.

Examples

The `ILendingPool` configured in `ProviderAave` (`0xB53C1a33016B2DC2fF3653530bfF1848a515c8c5` -> implementation: `0xc6845a5c768bf8d7681249f8927877efda425baf`)

code/contracts/mainnet/providers/ProviderAave.sol:L19-L21

```
function _getAaveProvider() internal pure returns (IAaveLendingPoolProvider) {
  return IAaveLendingPoolProvider(0xB53C1a33016B2DC2fF3653530bfF1848a515c8c5);
}
```

The `IAaveLendingPool` does not declare return values for any function, while upstream does.

code/contracts/interfaces/aave/IAaveLendingPool.sol:L1-L46

```
// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

interface IAaveLendingPool {
  function flashLoan(
    address receiverAddress,
    address[] calldata assets,
    uint256[] calldata amounts,
    uint256[] calldata modes,
    address onBehalfOf,
    bytes calldata params,
    uint16 referralCode
  ) external;

  function deposit(
    address _asset,
    uint256 _amount,
    address _onBehalfOf,
    uint16 _referralCode
  ) external;

  function withdraw(
    address _asset,
    uint256 _amount,
    address _to
  ) external;

  function borrow(
    address _asset,
    uint256 _amount,
    uint256 _interestRateMode,
    uint16 _referralCode,
    address _onBehalfOf
  ) external;

  function repay(
    address _asset,
    uint256 _amount,
    uint256 _rateMode,
    address _onBehalfOf
  ) external;

  function setUserUseReserveAsCollateral(address _asset, bool _useAsCollateral) external;
}
```

Methods: `withdraw()`, `repay()` return `uint256` in the original implementation for Aave, see:

https://etherscan.io/address/0xc6845a5c768bf8d7681249f8927877efda425baf#code

The `ILendingPool` configured for Geist:

Methods `withdraw()`, `repay()` return `uint256` in the original implementation for Geist, see:

https://ftmscan.com/address/0x3104ad2aadb6fe9df166948a5e3a547004862f90#code

Note: that the actual amount withdrawn does not necessarily need to match the `amount` provided with the function argument. Here's an excerpt of the upstream `LendingProvider.withdraw()`:

```
...
    if (amount == type(uint256).max) {
      amountToWithdraw = userBalance;
    }
...
   return amountToWithdraw;
```

And here's the code in Fuji that calls that method. This will break the `withdrawAll` functionality of `LendingProvider` if token `isFTM`.

code/contracts/fantom/providers/ProviderGeist.sol:L151-L165

```
function withdraw(address _asset, uint256 _amount) external payable override {
  IAaveLendingPool aave = IAaveLendingPool(_getAaveProvider().getLendingPool());

  bool isFtm = _asset == _getFtmAddr();
  address _tokenAddr = isFtm ? _getWftmAddr() : _asset;

  aave.withdraw(_tokenAddr, _amount, address(this));

  // convert WFTM to FTM
  if (isFtm)  {
    address unwrapper = _getUnwrapper();
    IERC20(_tokenAddr).univTransfer(payable(unwrapper), _amount);
    IUnwrapper(unwrapper).withdraw(_amount);
  }
}
```

Similar for `repay()`, which returns the actual amount repaid.

Recommendation

- Always use the original interface unless only a minimal subset of functions is used.
- Use the original upstream interfaces of the corresponding project (link via the respective npm packages if available).
- Avoid omitting parts of the function declaration! Especially when it comes to return values.

- Check return values. Use the value returned from `withdraw()` AND `repay()`

4.11 Missing slippage protection for rewards swap Medium

Description

In `FujiVaultFTM.harvestRewards` a swap transaction is generated using a call to `SwapperFTM.getSwapTransaction`. In all relevant scenarios, this call uses a minimum output amount of zero, which de-facto deactivates slippage checks. Most values from harvesting rewards can thus be siphoned off by sandwiching such calls.

Examples

`amountOutMin` is `0`, effectively disabling slippage control in the swap method.

code/contracts/fantom/SwapperFTM.sol:L49-L55

```
transaction.data = abi.encodeWithSelector(
  IUniswapV2Router01.swapExactETHForTokens.selector,
  0,
  path,
  msg.sender,
  type(uint256).max
);
```

Only success required

code/contracts/fantom/FujiVaultFTM.sol:L565-L567

```
// Swap rewards -> collateralAsset
(success, ) = swapTransaction.to.call{ value: swapTransaction.value }(swapTransaction.data);
require(success, "failed to swap rewards");
```

Recommendation

Use a slippage check such as for liquidator swaps:

code/contracts/fantom/FliquidatorFTM.sol:L476-L479

```
require(
  (priceDelta * SLIPPAGE_LIMIT_DENOMINATOR) / priceFromOracle < SLIPPAGE_LIMIT_NUMERATOR,
  Errors.VL_SWAP_SLIPPAGE_LIMIT_EXCEED
);
```

Or specify a non-zero `amountOutMin` argument in calls to `IUniswapV2Router01.swapExactETHForTokens`.

4.12 Unpredictable behavior due to admin front running or general bad timing Medium

Description

In several cases, the owner of deployed contracts can update or upgrade things in the system without warning. This has the potential to violate a security goal of the system.

Specifically, contract owners (a 2/3 EOA Gnosis Multisig) could use front running to make malicious changes just ahead of incoming transactions, or purely accidental adverse effects could occur due to unfortunate timing of changes.

Some instances of this are more important than others, but in general, users of the system should have assurances about the behavior of the action they're about to take.

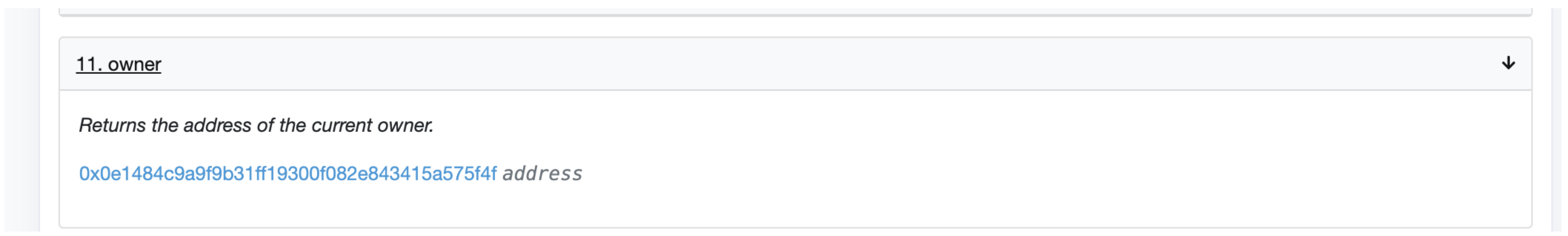
Examples

- `FujiAdmin`

The `owner` of `FujiAdmin` is `0x0e1484c9a9f9b31ff19300f082e843415a575f4f` and this address is a proxy to a `Gnosis Safe: Mastercopy 1.2.0` implementation, requiring 2/3 signatures to execute transactions. All three signees are EOA's.

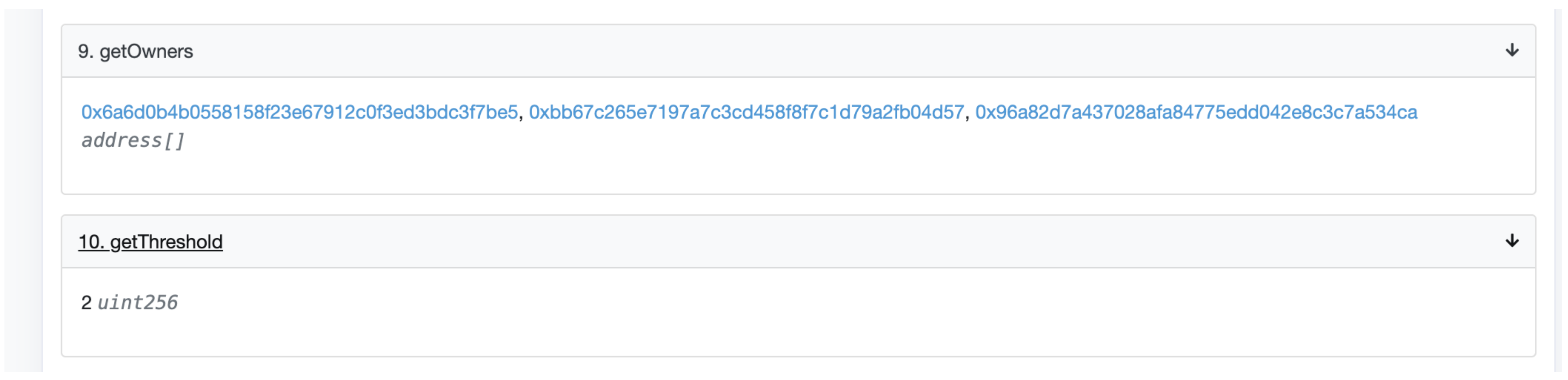
code/artifacts/1-core.deploy:L958-L960

```
"FujiAdmin": {
  "address": "0x4cB46032e2790D8CA10be6d0001e8c6362a76adA",
  "abi": [
```


11. owner

Returns the address of the current owner.

0x0e1484c9a9f9b31ff19300f082e843415a575f4f *address*


9. getOwners

0x6a6d0b4b0558158f23e67912c0f3ed3bdc3f7be5, 0xbb67c265e7197a7c3cd458f8f7c1d79a2fb04d57, 0x96a82d7a437028afa84775edd042e8c3c7a534ca *address[]*

10. getThreshold

2 *uint256*

- `Controller`, `FujiOracle`

The owner of `controller` seems to be a single EOA:

https://etherscan.io/address/0x3f366802F4e7576FC5DAA82890Cc6e04c85f3736#readContract

The owner of `FujiOracle` seems to be a single EOA:

https://etherscan.io/address/0xadF849079d415157CbBdb21BB7542b47077734A8#readContract

The owner of `FujiERC1155` seems to be a single EOA:

https://etherscan.io/address/0xa2d62f8b02225fbFA1cf8bF206C8106bDF4c692b#readProxyContract

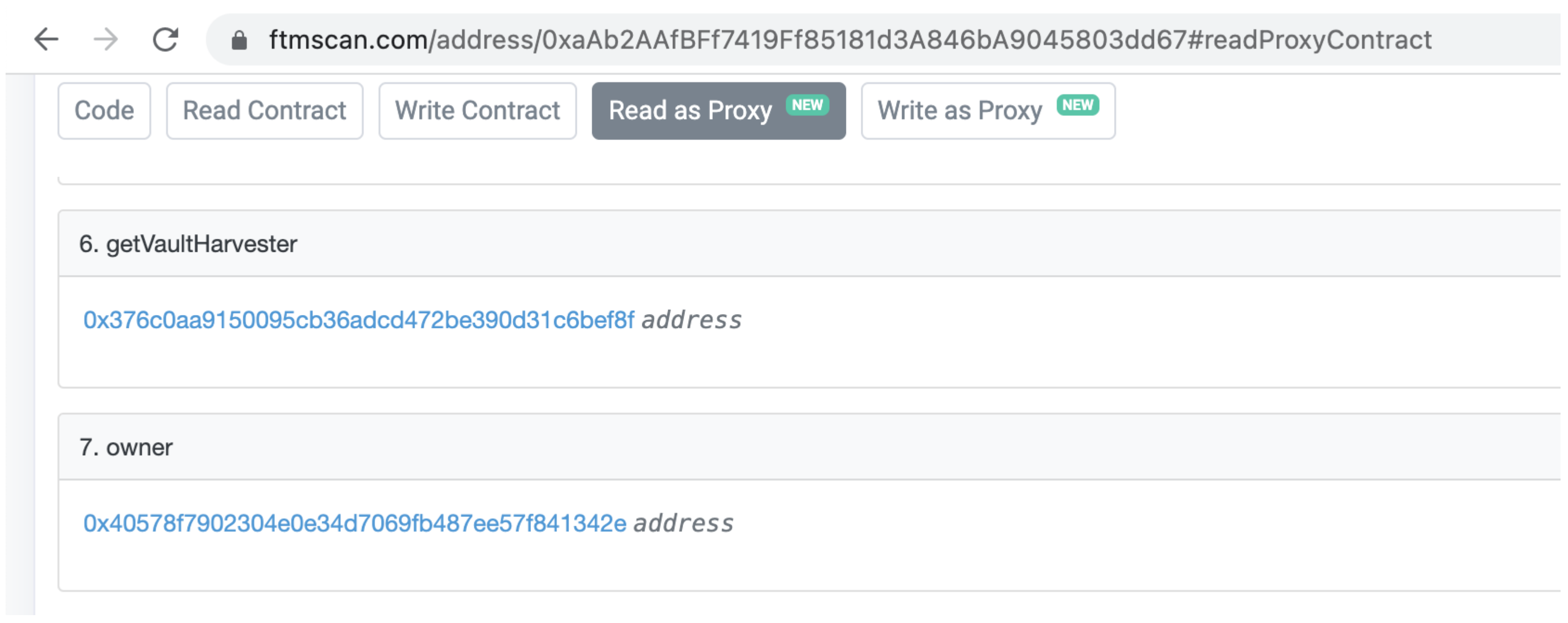
- `FujiAdmin` (fantom)

Deployer is 0xb98d4D4e205afF4d4755E9Df19BD0B8BD4e0f148 which is an EOA.

code/artifacts/250-core.deploy:L1-L5

```
{
  "FujiAdmin": {
    "address": "0xaAb2AAfBFf7419Ff85181d3A846bA9045803dd67",
    "deployer": "0xb98d4D4e205afF4d4755E9Df19BD0B8BD4e0f148",
    "abi": [
```

`FujiAdmin.owner` is 0x40578f7902304e0e34d7069fb487ee57f841342e which is a `GnosisSafeProxy`



ftmscan.com/address/0xaAb2AAfBFf7419Ff85181d3A846bA9045803dd67#readProxyContract
Code
Read Contract
Write Contract
Read as Proxy NEW
Write as Proxy NEW

6. getVaultHarvester

0x376c0aa9150095cb36adcd472be390d31c6bef8f *address*

7. owner

0x40578f7902304e0e34d7069fb487ee57f841342e *address*

Recommendation

The underlying issue is that users of the system can't be sure what the behavior of a function call will be, and this is because the behavior can change at any time.

We recommend giving the user advance notice of changes with a time lock. For example, all `onlyOwner` functionality requires two steps with a mandatory time window between them. The first step merely tells users that a particular change is coming, and the second step commits that change after a reasonable waiting period.

4.13 FujiOracle - `_getUSDPrice` does not detect stale oracle prices; General Oracle Risks Medium

Description

The external Chainlink oracle, which provides index price information to the system, introduces risk inherent to any dependency on third-party data sources. For example, the oracle could fall behind or otherwise fail to be maintained, resulting in outdated data being fed to the index price calculations. Oracle reliance has historically resulted in crippled on-chain systems, and complications that lead to these outcomes can arise from things as simple as network congestion.

This is more extreme in lesser-known tokens with fewer ChainLink Price feeds to update the price frequently.

Ensuring that unexpected oracle return values are correctly handled will reduce reliance on off-chain components and increase the resiliency of the smart contract system that depends on them.

The codebase, as is, relies on `chainLinkOracle.latestRoundData()` and does not check the `timestamp` or `answeredIn` round of the returned price.

Examples

- Here's how the oracle is consumed, skipping any fields that would allow checking for stale data:

code/contracts/FujiOracle.sol:L66-L77

```
/**
 * @dev Calculates the USD price of asset.
 * @param _asset: the asset address.
 * Returns the USD price of the given asset
 */
function _getUSDPrice(address _asset) internal view returns (uint256 price) {
  require(usdPriceFeeds[_asset] != address(0), Errors.ORACLE_NONE_PRICE_FEED);

  (, int256 latestPrice, , , ) = AggregatorV3Interface(usdPriceFeeds[_asset]).latestRoundData();

  price = uint256(latestPrice);
}
```

- Here's the implementation of the v0.6 FluxAggregator Chainlink feed with a note that timestamps should be checked.

contracts/src/v0.6/FluxAggregator.sol:L489-L490

```
* @return updatedAt is the timestamp when the round last was updated (i.e.
* answer was last computed)
```

Recommendation

Perform sanity checks on the price returned by the oracle. If the price is older, not within configured limits, revert or handle in other means.

The oracle does not provide any means to remove a potentially broken price-feed (e.g., by updating its address to `address(0)` or by pausing specific feeds or the complete oracle). The only way to pause an oracle right now is to deploy a new oracle contract. Therefore, consider adding minimally invasive functionality to pause the price-feeds if the oracle becomes unreliable.

Monitor the oracle data off-chain and intervene if it becomes unreliable.

On-chain, realistically, both `answeredInRound` and `updatedAt` must be checked within acceptable bounds.

- `answeredInRound == latestRound` - in this case, data may be assumed to be fresh while it might not be because the feed was entirely abandoned by nodes (no one starting a new round). Also, there's a good chance that many feeds won't always be super up-to-date (it might be acceptable to allow a threshold). A strict check might lead to transactions failing (race; e.g., round just timed out).
- `roundId + threshold >= answeredInRound` - would allow a deviation of threshold rounds. This check alone might still result in stale data to be used if there are no more rounds. Therefore, this should be combined with `updatedAt + threshold >= block.timestamp`.

4.14 Unclaimed or front-runnable proxy implementations Medium

Description

Various smart contracts in the system require initialization functions to be called. The point when these calls happen is up to the deploying address. Deployment and initialization in one transaction are typically safe, but it can potentially be front-run if the initialization is done in a separate transaction.

A frontrunner can call these functions to silently take over the contracts and provide malicious parameters or plant a backdoor during the deployment.

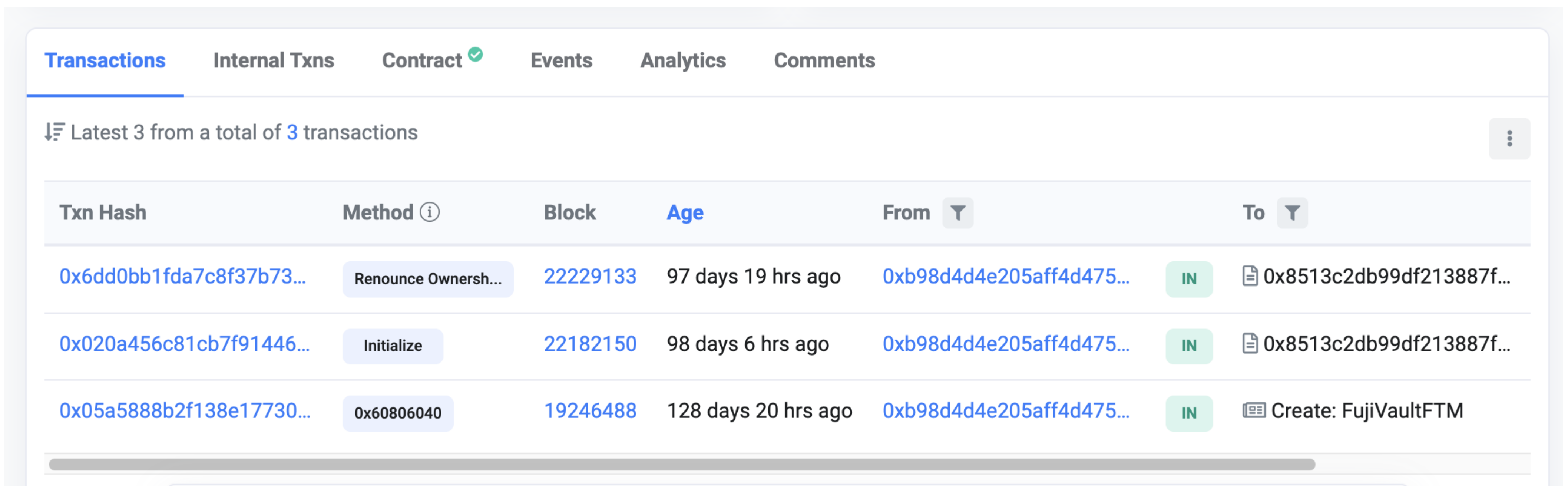
Leaving proxy implementations uninitialized further aides potential phishing attacks where users might claim that - just because a contract address is listed in the official documentation/code-repo - a contract is a legitimate component of the system. At the same time, it is 'only' a proxy implementation that an attacker claimed. For the end-user, it might be hard to distinguish whether this contract is part of the system or was a maliciously appropriated implementation.

Examples

code/contracts/mainnet/FujiVault.sol:L97-L102

```
function initialize(
  address _fujiadmin,
  address _oracle,
  address _collateralAsset,
  address _borrowAsset
) external initializer {
```

- `FujiVault` was initialized many days after deployment, and `FujiVault` inherits `VaultBaseUpgradeable`, which exposes a `delegatecall` that can be used to `selfdestruct` the contract's implementation.



Latest 3 from a total of 3 transactions

Txn Hash	Method	Block	Age	From		To
0x6dd0bb1fda7c8f37b73...	Renounce Ownersh...	22229133	97 days 19 hrs ago	0xb98d4d4e205aff4d475...	IN	0x8513c2db99df213887f...
0x020a456c81cb7f91446...	Initialize	22182150	98 days 6 hrs ago	0xb98d4d4e205aff4d475...	IN	0x8513c2db99df213887f...
0x05a5888b2f138e17730...	0x60806040	19246488	128 days 20 hrs ago	0xb98d4d4e205aff4d475...	IN	Create: FujiVaultFTM

Another `FujiVault` was deployed by `deployer` initialized in a 2-step approach that can theoretically silently be front-run.

code/artifacts/250-core.deploy:L2079-L2079

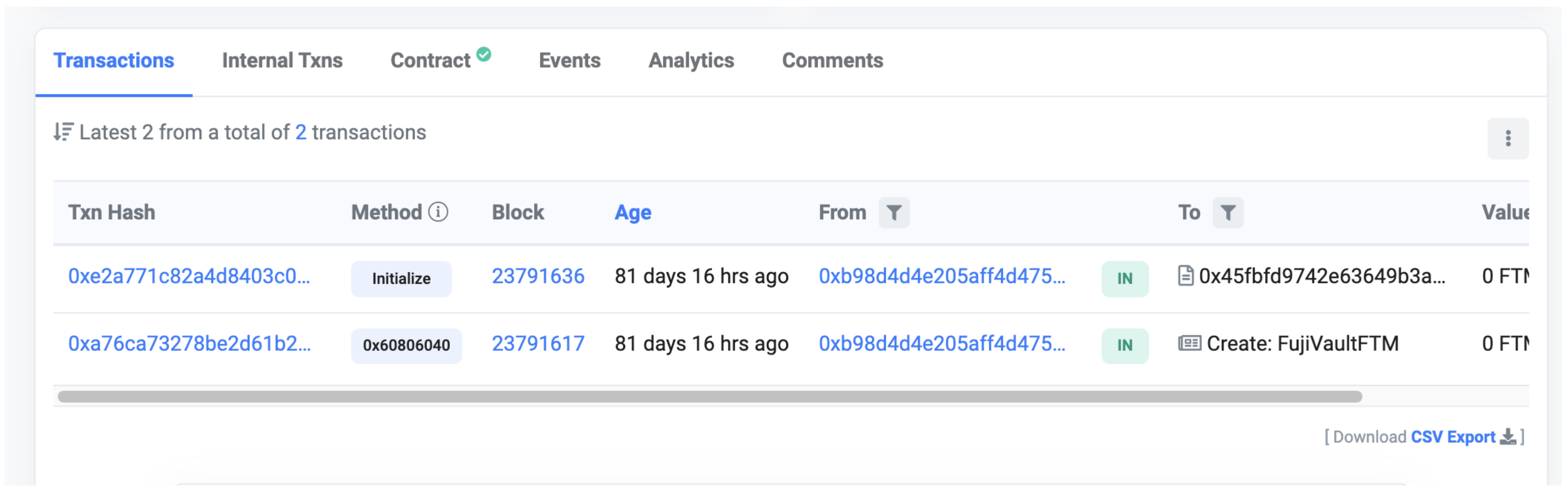
```
"deployer": "0xb98d4D4e205afF4d4755E9Df19BD0B8BD4e0f148",
```

Transactions of deployer:

https://ftmscan.com/txs?a=0xb98d4D4e205afF4d4755E9Df19BD0B8BD4e0f148&p=2

The specific contract was initialized **19** blocks after deployment.

https://ftmscan.com/address/0x8513c2db99df213887f63300b23c6dd31f1d14b0


Transactions | Internal Txns | Contract | Events | Analytics | Comments

Latest 2 from a total of 2 transactions

Txn Hash	Method	Block	Age	From		To	Value
0xe2a771c82a4d8403c0...	Initialize	23791636	81 days 16 hrs ago	0xb98d4d4e205aff4d475...	IN	0x45fbfd9742e63649b3a...	0 FTI
0xa76ca73278be2d61b2...	0x60806040	23791617	81 days 16 hrs ago	0xb98d4d4e205aff4d475...	IN	Create: FujiVaultFTM	0 FTI

[Download CSV Export]

- `FujiAdminFTM` (and others) don't seem to be initialized. (low prior; no risk other than pot. reputational damage)

code/artifacts/250-core.deploy:L1-L7

```
{
  "FujiAdmin": {
    "address": "0xaAb2AAfBFf7419Ff85181d3A846bA9045803dd67",
    "deployer": "0xb98d4D4e205afF4d4755E9Df19BD0B8BD4e0f148",
    "abi": [
      {
        "anonymous": false,
```

Recommendation

It is recommended to use constructors wherever possible to immediately initialize proxy implementations during deploy-time. The code is only run when the implementation is deployed and affects the proxy initializations. If other initialization functions are used, we recommend enforcing deployer access restrictions or a standardized, top-level `initialized` boolean, set to `true` on the first deployment and used to prevent future initialization.

Using constructors and locked-down initialization functions will significantly reduce potential developer errors and the possibility of attackers re-initializing vital system components.

4.15 Unused Import Minor

Description

The following dependency is imported but never used:

code/contracts/mainnet/flashloans/Flasher.sol:L13-L13

```
import "../../interfaces/IFujiMappings.sol";
```

Recommendation

Remove the unused import.

4.16 WFTM - Use of incorrect interface declarations Minor

Description

The `WFTMUnwrapper` and various providers utilize the `IWETH` interface declaration for handling funds denoted in `WFTM`. However, the WETH and WFTM implementations are different. `WFTM` returns `uint256` values to indicate error conditions while the `WETH` contract does not.

code/contracts/fantom/WFTMUnwrapper.sol:L7-L23

```
contract WFTMUnwrapper {
  address constant wftm = 0x21be370D5312f44cB42ce377BC9b8a0cEF1A4C83;

  receive() external payable {}

  /**
   * @notice Convert WFTM to FTM and transfer to msg.sender
   * @dev msg.sender needs to send WFTM before calling this withdraw
   * @param _amount amount to withdraw.
   */
  function withdraw(uint256 _amount) external {
    IWETH(wftm).withdraw(_amount);
    (bool sent, ) = msg.sender.call{ value: _amount }("");
    require(sent, "Failed to send FTM");
  }
}
```

The `WFTM` contract on Fantom returns an error return value. The error return value cannot be checked when utilizing the `IWETH` interface for `WFTM`. The error return values are never checked throughout the system for `WFTM` operations. This might be intentional to allow `amount=0` on `WETH` to act as a NOOP similar to `WETH`.

code/contracts/fantom/providers/ProviderGeist.sol:L115-L116

```
// convert FTM to WFTM
if (isFtm) IWETH(_tokenAddr).deposit{ value: _amount }();
```

Also see issues: issue 4.4, issue 4.5, issue 4.10

Recommendation

We recommend using the correct interfaces for all contracts instead of partial stubs. Do not modify the original function declarations, e.g., by omitting return value declarations. The codebase should also check return values where possible or explicitly state why values can safely be ignored in inline comments or the function's natspec documentation block.

4.17 Inconsistent `isFTM`, `isETH` checks Minor

Description

`LibUniversalERC20FTM.isFTM()` and `LibUniversalERC20.isETH()` identifies native assets by matching against two distinct addresses while some components only check for one.

Examples

The same is true for `FTM`.

- `Flasher` only identifies a native asset transfer by matching `asset` against `_ETH = 0xEeeeeEeeeEeEeeEeEeEeeEEEeeeeEeeeeeeeEEeE` while `univTransfer()` identifies it using `0x0 || 0xEeeeeEeeeEeEeeEeEeEeeEEEeeeeEeeeeeeeEEeE`

code/contracts/mainnet/flashloans/Flasher.sol:L122-L141

```
function callFunction(
  address sender,
  Account.Info calldata account,
  bytes calldata data
) external override {
  require(msg.sender == _dydxSoloMargin && sender == address(this), Errors.VL_NOT_AUTHORIZED);
  account;

  FlashLoan.Info memory info = abi.decode(data, (FlashLoan.Info));

  uint256 _value;
  if (info.asset == _ETH) {
    // Convert WETH to ETH and assign amount to be set as msg.value
    _convertWethToEth(info.amount);
    _value = info.amount;
  } else {
    // Transfer to Vault the flashloan Amount
    // _value is 0
    IERC20(info.asset).univTransfer(payable(info.vault), info.amount);
  }
```

- `LibUniversalERC20`

code/contracts/mainnet/libraries/LibUniversalERC20.sol:L8-L16

```
library LibUniversalERC20 {
  using SafeERC20 for IERC20;

  IERC20 private constant _ETH_ADDRESS = IERC20(0xEeeeeEeeeEeEeeEeEeEeeEEEeeeeEeeeeeeeEEeE);
  IERC20 private constant _ZERO_ADDRESS = IERC20(0x0000000000000000000000000000000000000000);

  function isETH(IERC20 token) internal pure returns (bool) {
    return (token == _ZERO_ADDRESS || token == _ETH_ADDRESS);
  }
```

code/contracts/mainnet/libraries/LibUniversalERC20.sol:L26-L40

```
function univTransfer(
  IERC20 token,
  address payable to,
  uint256 amount
) internal {
  if (amount > 0) {
    if (isETH(token)) {
      (bool sent, ) = to.call{ value: amount }("");
      require(sent, "Failed to send Ether");
    } else {
      token.safeTransfer(to, amount);
    }
  }
}
```

- There are multiple other instances of this

code/contracts/mainnet/Fliquidator.sol:L162-L162

```
uint256 _value = vAssets.borrowAsset == ETH ? debtTotal : 0;
```

Recommendation

Consider using a consistent way to identify native asset transfers (i.e. `ETH`, `FTM`) by using `LibUniversalERC20.isETH()`. Alternatively, the system can be greatly simplified by expecting WFTM and only working with it. This simplification will remove all special cases where the library must handle non-ERC20 interfaces.

4.18 FujiOracle - `setPriceFeed` should check asset and priceFeed decimals Minor

Description

`getPriceOf()` assumes that all price feeds return prices with identical decimals, but `setPriceFeed` does not enforce this. Potential misconfigurations can have severe effects on the system's internal accounting.

Examples

code/contracts/FujiOracle.sol:L27-L36

```
/**
 * @dev Sets '_priceFeed' address for a '_asset'.
 * Can only be called by the contract owner.
 * Emits a {AssetPriceFeedChanged} event.
 */
function setPriceFeed(address _asset, address _priceFeed) public onlyOwner {
  require(_priceFeed != address(0), Errors.VL_ZERO_ADDR);
  usdPriceFeeds[_asset] = _priceFeed;
  emit AssetPriceFeedChanged(_asset, _priceFeed);
}
```

Recommendation

We recommend adding additional checks to detect unexpected changes in assets' properties. Safeguard price feeds by enforcing `priceFeed == address(0) || priceFeed.decimals() == 8`. This allows the owner to disable a `priceFeed` (setting it to zero) and otherwise ensure that the feed is compatible and indeed returns `8` decimals.

4.19 Unchecked function return values for low-level calls

Description

It should be noted that the swapping and harvesting transactions sometimes return values to the function caller. While the low-level call is checked for "success", the return values are not actively handled. This can be intentional but should be verified.

Before calling the external contract, there is no check whether a contract is deployed at that address. Since destinations seem to be hardcoded in the Swapper/Harvester modules, we assume this has been ensured before deploying the contract. However, we suggest checking that code is deployed at the destination address, especially for upgradeable contracts.

We raise this as an informational finding as both the Harvester and Swapper flows using `token.balanceOf(this)`, which might make this check obsolete. However, potential future third-party Swapper/Harvester additions to the protocol might return error codes that need to be checked for.

Examples

- Geist/Uniswap and `WFTM` methods may return amounts or error codes

code/contracts/fantom/FujiVaultFTM.sol:L549-L551

```
// Claim rewards
(bool success, ) = harvestTransaction.to.call(harvestTransaction.data);
require(success, "failed to harvest rewards");
```

code/contracts/fantom/FujiVaultFTM.sol:L565-L567

```
// Swap rewards -> collateralAsset
(success, ) = swapTransaction.to.call{ value: swapTransaction.value }(swapTransaction.data);
require(success, "failed to swap rewards");
```

4.20 Use the compiler to resolve function selectors for interfaces

Description

Function signatures of known contract and interface types are available to the compiler. We recommend using `abi.encodeWithSelector(IProvider.withdraw.selector, ...)` instead of the more error prone `abi.encodeWithSignature("withdraw(address,uint256)", ...)` equivalent. Using the former method avoids hard-to-detect errors stemming from typos, interface changes, etc.

Examples

code/contracts/abstracts/vault/VaultBaseUpgradeable.sol:L57-L84

```
/**
 * @dev Executes withdraw operation with delegatecall.
 * @param _amount: amount to be withdrawn
 * @param _provider: address of provider to be used
 */
function _withdraw(uint256 _amount, address _provider) internal {
  bytes memory data = abi.encodeWithSignature(
    "withdraw(address,uint256)",
    vAssets.collateralAsset,
    _amount
  );
  _execute(_provider, data);
}

/**
 * @dev Executes borrow operation with delegatecall.
 * @param _amount: amount to be borrowed
 * @param _provider: address of provider to be used
 */
function _borrow(uint256 _amount, address _provider) internal {
  bytes memory data = abi.encodeWithSignature(
    "borrow(address,uint256)",
    vAssets.borrowAsset,
    _amount
  );
  _execute(_provider, data);
}
```

4.21 Reduce code complexity

Description

Throughout the codebase, snippets of code and whole functions have been copy-pasted. This duplication significantly increases code complexity and the potential for bugs. We recommend re-using code across modules or providing library contracts that implement re-usable code fragments.

Examples

- Providers should use `LibUniversalERC20FTM.isFTM` instead of re-implementing `Helper.isFTM`.

code/contracts/fantom/providers/ProviderCream.sol:L17-L19

```
function _isFTM(address token) internal pure returns (bool) {
  return (token == address(0) || token == address(0xFFfFfFffFFfffFFfFFfFFFFFffFFFffffFfFFFfF));
}
```

code/contracts/fantom/providers/ProviderScream.sol:L17-L19

```
function _isFTM(address token) internal pure returns (bool) {
  return (token == address(0) || token == address(0xFFfFfFffFFfffFFfFFfFFFFFffFFFffffFfFFFfF));
}
```

- `ProviderGeist` should provide an internal method instead of implementing multiple variants of the `isFtm` to token address mapping. E.g., both calls do the same thing. They select a different return value from the external call. Avoid re-implementing an inconsistent `isFtm` variant. Require that `isFtm && amount != 0` on `deposit/payback`.

code/contracts/fantom/providers/ProviderGeist.sol:L57-L67

```
function getBorrowBalance(address _asset) external view override returns (uint256) {
  IAaveDataProvider aaveData = _getAaveDataProvider();

  bool isFtm = _asset == _getFtmAddr();
  address _tokenAddr = isFtm ? _getWftmAddr() : _asset;

  (, , uint256 variableDebt, , , , , , ) = aaveData.getUserReserveData(_tokenAddr, msg.sender);

  return variableDebt;
}
```

code/contracts/fantom/providers/ProviderGeist.sol:L43-L52

```
function getBorrowRateFor(address _asset) external view override returns (uint256) {
  IAaveDataProvider aaveData = _getAaveDataProvider();

  (, , , , uint256 variableBorrowRate, , , , , ) = IAaveDataProvider(aaveData).getReserveData(
    _asset == _getFtmAddr() ? _getWftmAddr() : _asset
  );

  return variableBorrowRate;
}
```

Also, note the unnecessary double cast `IAaveDataProvider`.

code/contracts/fantom/providers/ProviderGeist.sol:L73-L87

```
function getBorrowBalanceOf(address _asset, address _who)
  external
  view
  override
  returns (uint256)
{
  IAaveDataProvider aaveData = _getAaveDataProvider();

  bool isFtm = _asset == _getFtmAddr();
  address _tokenAddr = isFtm ? _getWftmAddr() : _asset;

  (, , uint256 variableDebt, , , , , , ) = aaveData.getUserReserveData(_tokenAddr, _who);

  return variableDebt;
}
```

- Consider removing support for the native currency altogether in favor of only accepting pre-wrapped `WFTM` (`WETH`). This should remove a lot of glue code currently implemented to auto-wrap/unwrap native currency.
- Unused functionality

code/contracts/fantom/providers/ProviderCream.sol:L52-L57

```
function _exitCollatMarket(address _cyTokenAddress) internal {
  // Create a reference to the corresponding network Comptroller
  IComptroller comptroller = IComptroller(_getComptrollerAddress());

  comptroller.exitMarket(_cyTokenAddress);
}
```

4.22 Unusable state variable in dYdX provider

Description

Remove the state variable `donothing` . Providers are always called via staticcall or delegatecall and should not hold any state.

code/contracts/mainnet/providers/ProviderDYDX.sol:L93-L95

```

bool public donothing = true;
```

4.23 Use enums instead of hardcoded integer literals

Description

Hardcoded integers are used throughout the codebase to denote states and distinguish between states. The code's complexity can be significantly reduced by using descriptive enum values.

Examples

- `2` should be `InterestRateMode.VARIABLE`

code/contracts/fantom/providers/ProviderGeist.sol:L184-L184

```
aave.repay(_tokenAddr, _amount, 2, address(this));
```

code/contracts/fantom/providers/ProviderGeist.sol:L136-L136

```
aave.borrow(_tokenAddr, _amount, 2, 0, address(this));
```

- `_farmProtocolNum` and `harvestType` should be refactored to their enum equivalents:

code/contracts/mainnet/Harvester.sol:L20-L32

```
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[]));
```

- label the flashloan providers with an enum representing their name

code/contracts/fantom/flashloans/FlasherFTM.sol:L72-L78

```
if (_flashnum == 0) {
  _initiateGeistFlashLoan(info);
} else if (_flashnum == 2) {
  _initiateCreamFlashLoan(info);
} else {
  revert(Errors.VL_INVALID_FLASH_NUMBER);
}
```

4.24 Redundant harvest check in vault

Description

In the `FujiVaultFTM.harvestRewards` function, the check for a returned token's address in the if condition and `require` statement overlap with `tokenReturned != address(0)`.

Examples

code/contracts/mainnet/FujiVault.sol:L553-L555

```
if (tokenReturned != address(0)) {
  uint256 tokenBal = IERC20Upgradeable(tokenReturned).univBalanceOf(address(this));
  require(tokenReturned != address(0) && tokenBal > 0, Errors.VL_HARVESTING_FAILED);
```

Recommendation

We recommend removing one of the statements for gas savings and increased readability.

4.25 Redundant use of `immutable` for constants

Description

The `FlasherFTM` contract declares `immutable` state variables even though they are never set in the constructor. Consider declaring them as `constant` instead unless they are to be set on construction time. See the Solidity Documentation for further details:

> [...] For constant variables, the value has to be fixed at compile-time, while for immutable, it can still be assigned at construction time. [...]

Examples

code/contracts/mainnet/flashloans/Flasher.sol:L37-L44

```
address private immutable _aaveLendingPool = 0x7d2768dE32b0b80b7a3454c06BdAc94A69DDc7A9;
address private immutable _dydxSoloMargin = 0x1E0447b19BB6EcFdAe1e4AE1694b0C3659614e4e;

// IronBank
address private immutable _cyFlashloanLender = 0x1a21Ab52d1Ca1312232a72f4cf4389361A479829;
address private immutable _cyComptroller = 0xAB1c342C7bf5Ec5F02ADEA1c2270670bCa144CbB;

// need to be payable because of the conversion ETH <> WETH
```

code/contracts/fantom/flashloans/FlasherFTM.sol:L36-L39

```
address private immutable _geistLendingPool = 0x9FAD24f572045c7869117160A571B2e50b10d068;
IFujiMappings private immutable _crMappings =
  IFujiMappings(0x1eEdE44b91750933C96d2125b6757C4F89e63E20);
```

4.26 Redeclaration of constant values in multiple contracts

Description

Throughout the codebase, constant values are redeclared in various contracts. This duplication makes the code harder to maintain and increases the risk for bugs. A central contract, e.g., `Constants.sol`, `ConstantsFTM.sol`, and `ConstantsETH.sol`, to declare the constants used throughout the codebase instead of redeclaring them in multiple source units can fix this issue. Ideally, for example, an address constant for an external component is only configured in a single place but consumed by multiple contracts. This will significantly reduce the potential for misconfiguration.

Avoid hardcoded addresses and use meaningful, constant names for them.

Note that the solidity compiler is going to inline constants where possible.

Examples

code/contracts/mainnet/WETHUnwrapper.sol:L7-L9

```
contract WETHUnwrapper {
  address constant weth = 0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2;
```

code/contracts/mainnet/Swapper.sol:L16-L19

```
address public constant ETH = 0xEeeeeEeeeEeEeeEeEeEeeEEEeeeeEeeeeeeeEEeE;
address public constant WETH = 0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2;
address public constant SUSHI_ROUTER_ADDR = 0xd9e1cE17f2641f24aE83637ab66a2cca9C378B9F;
```

code/contracts/mainnet/FujiVault.sol:L32-L34

```
address public constant ETH = 0xEeeeeEeeeEeEeeEeEeEeeEEEeeeeEeeeeeeeEEeE;
```

code/contracts/mainnet/Fliquidator.sol:L31-L31

```
address public constant ETH = 0xEeeeeEeeeEeEeeEeEeEeeEEEeeeeEeeeeeeeEEeE;
```

code/contracts/mainnet/providers/ProviderCompound.sol:L14-L18

```
contract HelperFunct {
  function _isETH(address token) internal pure returns (bool) {
    return (token == address(0) || token == address(0xEeeeeEeeeEeEeeEeEeEeeEEEeeeeEeeeeeeeEEeE));
  }
```

code/contracts/mainnet/libraries/LibUniversalERC20.sol:L10-L14

```
IERC20 private constant _ETH_ADDRESS = IERC20(0xEeeeeEeeeEeEeeEeEeEeeEEEeeeeEeeeeeeeEEeE);
IERC20 private constant _ZERO_ADDRESS = IERC20(0x0000000000000000000000000000000000000000);

function isETH(IERC20 token) internal pure returns (bool) {
```

code/contracts/mainnet/flashloans/Flasher.sol:L34-L36

```
address private constant _ETH = 0xEeeeeEeeeEeEeeEeEeEeeEEEeeeeEeeeeeeeEEeE;
address private constant _WETH = 0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2;
```

- Use meaningful names instead of hardcoded addresses

code/contracts/mainnet/Harvester.sol:L20-L29

```
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));
```

- Avoid unnamed hardcoded inlined addresses

code/contracts/fantom/providers/ProviderCream.sol:L157-L162

```
if (_isFTM(_asset)) {
  // Transform FTM to WFTM
  IWETH(0x21be370D5312f44cB42ce377BC9b8a0cEF1A4C83).deposit{ value: _amount }();
  _asset = address(0x21be370D5312f44cB42ce377BC9b8a0cEF1A4C83);
}
```

- comptroller address - can also be `private constant` state variables as the compiler/preprocessor will inline them.

code/contracts/fantom/providers/ProviderCream.sol:L21-L31

```
function _getMappingAddr() internal pure returns (address) {
  return 0x1eEdE44b91750933C96d2125b6757C4F89e63E20; // Cream fantom mapper
}

function _getComptrollerAddress() internal pure returns (address) {
  return 0x4250A6D3BD57455d7C6821eECb6206F507576cD2; // Cream fantom
}

function _getUnwrapper() internal pure returns(address) {
  return 0xee94A39D185329d8c46dEA726E01F91641E57346;
}
```

- `WFTM` multiple re-declarations

code/contracts/fantom/WFTMUnwrapper.sol:L7-L9

```
contract WFTMUnwrapper {
  address constant wftm = 0x21be370D5312f44cB42ce377BC9b8a0cEF1A4C83;
```

code/contracts/fantom/providers/ProviderGeist.sol:L27-L29

```
function _getWftmAddr() internal pure returns (address) {
  return 0x21be370D5312f44cB42ce377BC9b8a0cEF1A4C83;
}
```

code/contracts/fantom/providers/ProviderCream.sol:L79-L81

```
  IWETH(0x21be370D5312f44cB42ce377BC9b8a0cEF1A4C83).deposit{ value: _amount }();
  _asset = address(0x21be370D5312f44cB42ce377BC9b8a0cEF1A4C83);
}
```

4.27 Always use the best available type

Description

Declare state variables with the best type available and downcast to `address` if needed. Typecasting inside the corpus of a function is unneeded when the parameter's type is known beforehand. Declare the best type in function arguments, state vars. Always return the best type available instead of falling back to `address`.

Examples

There are many more instances of this, but here's a list of samples:

- Should be declared with the correct types/interfaces instead of `address`

code/contracts/FujiAdmin.sol:L14-L20

```

address private _flasher;
address private _fliquidator;
address payable private _ftreasury;
address private _controller;
address private _vaultHarvester;
```

- Should return the correct type/interfaces instead of `address`

code/contracts/FujiAdmin.sol:L144-L147

```
 */
function getSwapper() external view override returns (address) {
  return _swapper;
}
```

- Should declare the argument with the correct type instead of casting in the function body.

code/contracts/Controller.sol:L73-L80

```
function doRefinancing(
  address _vaultAddr,
  address _newProvider,
  uint8 _flashNum
) external isValidVault(_vaultAddr) onlyOwnerOrExecutor {

  IVault vault = IVault(_vaultAddr);
```

- Should make the `FujiVaultFTM.fujiERC1155` state variable of type `IFujiERC1155`

code/contracts/fantom/FujiVaultFTM.sol:L438-L445

```
IFujiERC1155(fujiERC1155).updateState(
  vAssets.borrowID,
  IProvider(activeProvider).getBorrowBalance(vAssets.borrowAsset)
);
IFujiERC1155(fujiERC1155).updateState(
  vAssets.collateralID,
  IProvider(activeProvider).getDepositBalance(vAssets.collateralAsset)
);
```

- Return the best type available

code/contracts/fantom/providers/ProviderCream.sol:L25-L31

```
function _getComptrollerAddress() internal pure returns (address) {
  return 0x4250A6D3BD57455d7C6821eECb6206F507576cD2; // Cream fantom
}

function _getUnwrapper() internal pure returns(address) {
  return 0xee94A39D185329d8c46dEA726E01F91641E57346;
}
```

Appendix 1 - Files in Scope

This audit covered the following files:

File	SHA-1 hash
`./contracts/Controller.sol`	`ff7ac267bc08adeb710e7ab78b8d9b8818276f80`
`./contracts/fantom/flashloans/FlasherFTM.sol`	`5da4d3a4b796cd63255bf652d924b0a2fcfc9058`
`./contracts/fantom/FliquidatorFTM.sol`	`593c2f2d376fd9e7fdcf4ac7b6478a36e6830861`
`./contracts/fantom/FujiVaultFTM.sol`	`d360a1c97047cc14b91f13a24fe8277c652965aa`
`./contracts/fantom/libraries/LibUniversalERC20FTM.sol`	`33d1c1e58b19ef03d2d19177097b9fe3b66db166`

File	SHA-1 hash
`./contracts/fantom/libraries/LibUniversalERC20UpgradeableFTM.sol`	`be61be7ae35ddfe726b2c29915aa8329604d5a75`
`./contracts/fantom/providers/ProviderCream.sol`	`b3c8e2c67f87684e10d9282b19475d392c2788e3`
`./contracts/fantom/providers/ProviderGeist.sol`	`5ece35982f5a92d822df9d1c44c1b10fb3ab6e65`
`./contracts/fantom/providers/ProviderScream.sol`	`cde8e0390b40e75e5373aadee9b12b91e7805cc0`
`./contracts/fantom/SwapperFTM.sol`	`5eb0e09f210a6b2dc0dc44debcd06338ddc379f1`
`./contracts/fantom/WFTMUnwrapper.sol`	`b8d5842f26b140e32cb28425c1bd779366816359`
`./contracts/FujiAdmin.sol`	`4540517cd55d2da08ac08124e25ce86cfe8cd46b`
`./contracts/FujiERC1155.sol`	`4e952b95cb0a242afcf4db09bd68718b7c963ccf`
`./contracts/FujiMapping.sol`	`021239e02a9bb2079616c85d71e032ecae709bc4`
`./contracts/FujiOracle.sol`	`e0112d74aa3881da1704b9efcba62377deb9842f`

Appendix 2 - Disclosure

ConsenSys Diligence (“CD”) typically receives compensation from one or more clients (the “Clients”) for performing the analysis contained in these reports (the “Reports”). The Reports may be distributed through other means, including via ConsenSys publications and other distributions.

The Reports are not an endorsement or indictment of any particular project or team, and the Reports do not guarantee the security of any particular project. This Report does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. No Report provides any warranty or representation to any Third-Party in any respect, including regarding the bugfree nature of code, the business model or proprietors of any such business model, and the legal compliance of any such business. No third party should rely on the Reports in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset. Specifically, for the avoidance of doubt, this Report does not constitute investment advice, is not intended to be relied upon as investment advice, is not an endorsement of this project or team, and it is not a guarantee as to the absolute security of the project. CD owes no duty to any Third-Party by virtue of publishing these Reports.

PURPOSE OF REPORTS The Reports and the analysis described therein are created solely for Clients and published with their consent. The scope of our review is limited to a review of code and only the code we note as being within the scope of our review within this report. Any Solidity code itself presents unique and unquantifiable risks as the Solidity language itself remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer, or any other areas beyond specified code that could present security risks. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. In some instances, we may perform penetration testing or infrastructure assessments depending on the scope of the particular engagement.

CD makes the Reports available to parties other than the Clients (i.e., “third parties”) – on its website. CD hopes that by making these analyses publicly available, it can help the blockchain ecosystem develop technical best practices in this rapidly evolving area of innovation.

LINKS TO OTHER WEB SITES FROM THIS WEB SITE You may, through hypertext or other computer links, gain access to web sites operated by persons other than ConsenSys and CD. Such hyperlinks are provided for your reference and convenience only, and are the exclusive responsibility of such web sites’ owners. You agree that ConsenSys and CD are not responsible for the content or operation of such Web sites, and that ConsenSys and CD shall have no liability to you or any other person or entity for the use of third party Web sites. Except as described below, a hyperlink from this web Site to another web site does not imply or mean that ConsenSys and CD endorses the content on that Web site or the operator or operations of that site. You are solely responsible for determining the extent to which you may use any content at any other web sites to which you link from the Reports. ConsenSys and CD assumes no responsibility for the use of third party software on the Web Site and shall have no liability whatsoever to any person or entity for the accuracy or completeness of any outcome generated by such software.

TIMELINESS OF CONTENT The content contained in the Reports is current as of the date appearing on the Report and is subject to change without notice. Unless indicated otherwise, by ConsenSys and CD.