FortiFi Post-Audit Change Report

Prepared: 4/14/2024

Audit Commit: https://github.com/0xFortiFi/FortiFi-Vaults/commit/3c6a0709427caa12ce6427965be34b20cbdcbb2c

Current Commit: https://github.com/0xFortiFi/FortiFi-Vaults/commit/0228206f0be76b0661ddf43b328c6adcfeb8560c

This report seeks to highlight changes to the FortiFi smart contracts that have occurred after the audit by Blaize Security. You can find a copy of the Blaize Security audit as well as their test suite here:

https://github.com/0xFortiFi/FortiFi-Vaults/tree/main/blaize-audit

FortiFiSAMSVault

1. Addition of ‘strategyIsBricked’ mapping

   Problem: It was determined that users may be unable to withdraw their deposits if one of the underlying strategies became broken (i.e. when Platypus was exploited, users who deposited into certain pools on Yield Yak were unable to withdraw from the pool).

   Solution: Allow the setting of a specific strategy to become ‘bricked’, in which case the protocol will simply send the user the receipt tokens for that specific underlying strategy instead of withdrawing and returning deposit tokens.

   Implementation:

   a) A new custom error was created to prevent deposits while one of the strategies is bricked:

```
function _deposit(uint256 _amount, uint256 _tokenId, bool _isAdd) internal {
    TokenInfo storage _info = tokenInfo[_tokenId];
    uint256 _remainder = _amount;

    uint256 _length = strategies.length;
    for (uint256 i = 0; i < _length; i++) {
        Strategy memory _strategy = strategies[i];

        if (strategyIsBricked[_strategy.strategy]) revert StrategyBricked();

        // cannot add to position if strategies have changed. must rebalance first
        if (_isAdd) {
            if (_strategy.strategy != _info.positions[i].strategy.strategy) revert C
        }
```

   b) A new condition was added to the _withdraw function:

```
/// @notice Internal withdraw function that withdraws from strategies and calculates profits.
function _withdraw(uint256 _tokenId) internal returns(uint256 _proceeds, uint256 _profit) {
    TokenInfo memory _info = tokenInfo[_tokenId];
    uint256 _length = _info.positions.length;

    for (uint256 i = 0 ; i < _length; i++) {
        IStrategy _strat = IStrategy(_info.positions[i].strategy.strategy);
        bool _bricked = strategyIsBricked[_info.positions[i].strategy.strategy];

        if (_bricked) {
            // send receipt tokens to user without withdrawing from strategy
            IERC20(_info.positions[i].strategy.strategy).safeTransfer(msg.sender, _info.positions[i].receipt);
        } else if (_info.positions[i].strategy.isFortiFi) {
            _strat.withdrawFromFortress(_info.positions[i].receipt, msg.sender, _tokenId);
        } else {
            _strat.withdraw(_info.positions[i].receipt);
        }
    }
```

c) A new setter was added to set strategies as bricked or !bricked:

```
/// @notice Function to set a strategy as bricked
function setStrategyAsBricked(address _strategy, bool _bool) external onlyOwner {
    strategyIsBricked[_strategy] = _bool;
}
```

2. Change to tokenInfo mapping

Problem: Default getter on public mappings for structs does not return data as expected.

Solution: tokenInfo mapping was made private and custom getter introduced.

Implementation:

```
/// @notice View function that returns tokenInfo
function getTokenInfo(uint256 _tokenId) public view returns(TokenInfo memory) {
    return tokenInfo[_tokenId];
}
```

FortiFiMASSVaultV2

This contract was marked V2 as it required the removal of some functionality, namely the ability to utilize SAMS vaults as an underlying strategy as well as the setBpsForStrategies function. This was required to add the strategyIsBricked logic because the contract size was too large to compile. You can the V1 contract here: FortiFiMASSVault.

1. Removal of support for SAMS vaults

   Problem: The contract size was already too big, and gas constraints made it unlikely that utilize a SAMS vault as one of many underlying strategies would be feasible.

   Solution: Remove support for SAMS, including ERC1155Receiver inheritance.

   Note: isSAMS was left in the Strategy struct in order to minimize downstream effects even though it will always be false.

   Implementation:

   a) Removed isSAMS condition from _deposit logic:

```
function _deposit(uint256 _amount, uint256 _tokenId, bool _isAdd) internal {
    for (uint256 i = 0; i < _length; i++) {

        if (_isSAMS) {
            if (_isAdd) {
                _addSAMS(_depositAmount, _strategy.strategy, _info.positions[i].receipt);
            } else {
                // if position is new, deposit and push to positions
                _receiptToken = _depositSAMS(_depositAmount, _strategy.strategy);
                _info.positions.push(Position({strategy: _strategy, receipt: _receiptToken}));
            }
        } else {
```

   b) Removed SAMS-specific internal functions:

```
/// @notice internal function to deposit to SAMS vault
function _depositSAMS(uint256 _amount, address _strategy) internal returns (uint256 _receiptToken) {
    ISAMS _sams = ISAMS(_strategy);
    ISAMS.TokenInfo memory _receiptInfo;

    (_receiptToken, _receiptInfo) = _sams.deposit(_amount);
}

/// @notice internal function to add to SAMS vault
function _addSAMS(uint256 _amount, address _strategy, uint256 _tokenId) internal {
    ISAMS _sams = ISAMS(_strategy);
    ISAMS.TokenInfo memory _receiptInfo;

    _receiptInfo = _sams.add(_amount, _tokenId);
}
```

c) Removed isSAMS condition in _withdraw logic

```
function _withdraw(uint256 _tokenId) internal returns(uint256 _proceeds, uint256 _profit) {
    uint256 _length = _info.positions.length;
    _proceeds = 0;

    for (uint256 i = 0 ; i < _length; i++) {
        // withdraw based on the type of underlying strategy, if not SAMS check if FortiFi strategy
        if (_info.positions[i].strategy.isSAMS) {
            ISAMS _strat = ISAMS(_info.positions[i].strategy.strategy);
            _strat.withdraw(_info.positions[i].receipt);
        } else {
```

d) Removed ERC1155Receiver functions:

```
function onERC1155Received(
    address,
    address,
    uint256,
    uint256,
    bytes memory
) public virtual override returns (bytes4) {
    return this.onERC1155Received.selector;
}

function onERC1155BatchReceived(
    address,
    address,
    uint256[] memory,
    uint256[] memory,
    bytes memory
) public virtual override returns (bytes4) {
    return this.onERC1155BatchReceived.selector;
}
```

2. Modification of oracle requirements

   Problem: Original logic assumed that decimal places for the oracle would match the decimals of the token for which price was provided. This turned out to be false in all cases except BTC.b, because it is standard for oracles to use 8 decimals when returning a price denominated in USD.

   Solution: Implement flexible FortiFiPriceOracle contracts and ensure oracles only use 8 decimal places for the price returned.

   Implementation:

```
if (_strategies[i].depositToken != depositToken &&
    (_strategies[i].oracle == address(0) ||
     _strategies[i].depositToken != IFortiFiPriceOracle(_strategies[i].oracle).token() ||
     _strategies[i].decimals != IFortiFiPriceOracle(_strategies[i].oracle).decimals())
   ) revert InvalidOracle();
if (_strategies[i].decimals <= USDC_DECIMALS &&
    _strategies[i].depositToken != depositToken) revert InvalidDecimals();
```

```
if (_strategies[i].depositToken != depositToken &&
    (_strategies[i].oracle == address(0) ||
     _strategies[i].depositToken != IFortiFiPriceOracle(_strategies[i].oracle).token() ||
     IFortiFiPriceOracle(_strategies[i].oracle).decimals() != 8)
   ) revert InvalidOracle();
```

3. Modify swap calculation for new oracle specifications

   Problem: New oracle specifications (see above) made existing swap logic incorrect.

   Solution: Modify swap logic to account for oracles returning prices with 8 decimals

   Implementation:

```
uint256 _swapAmount = _amount * (10**_strat.decimals) / _latestPrice*10**(8 - USDC_DECIMALS);
```

   NOTE Similar changes made in all swap functions

4. Addition of 'strategyIsBricked' mapping

   Problem: It was determined that users may be unable to withdraw their deposits if one of the underlying strategies became broken (i.e. when Platypus was exploited, users who deposited into certain pools on Yield Yak were unable to withdraw from the pool).

Solution: Allow the setting of a specific strategy to become 'bricked', in which case the protocol will simply send the user the receipt tokens for that specific underlying strategy instead of withdrawing and returning deposit tokens.

Implementation: Same as for SAMS (see above)

5. Change to tokenInfo mapping

   Problem: Default getter on public mappings for structs does not return data as expected.

   Solution: tokenInfo mapping was made private and custom getter introduced.

   Implementation: Same as for SAMS (see above)

6. Reduce minimum slippage

   Problem: The minimum slippage allowed by the contract was 1%, but with the use of new routers like the FortiFiLBRouter swaps could be made with far less impact, particularly on stablecoin swaps.

   Solution: Reduce the minimum slippage to 0.1%

   Implementation:

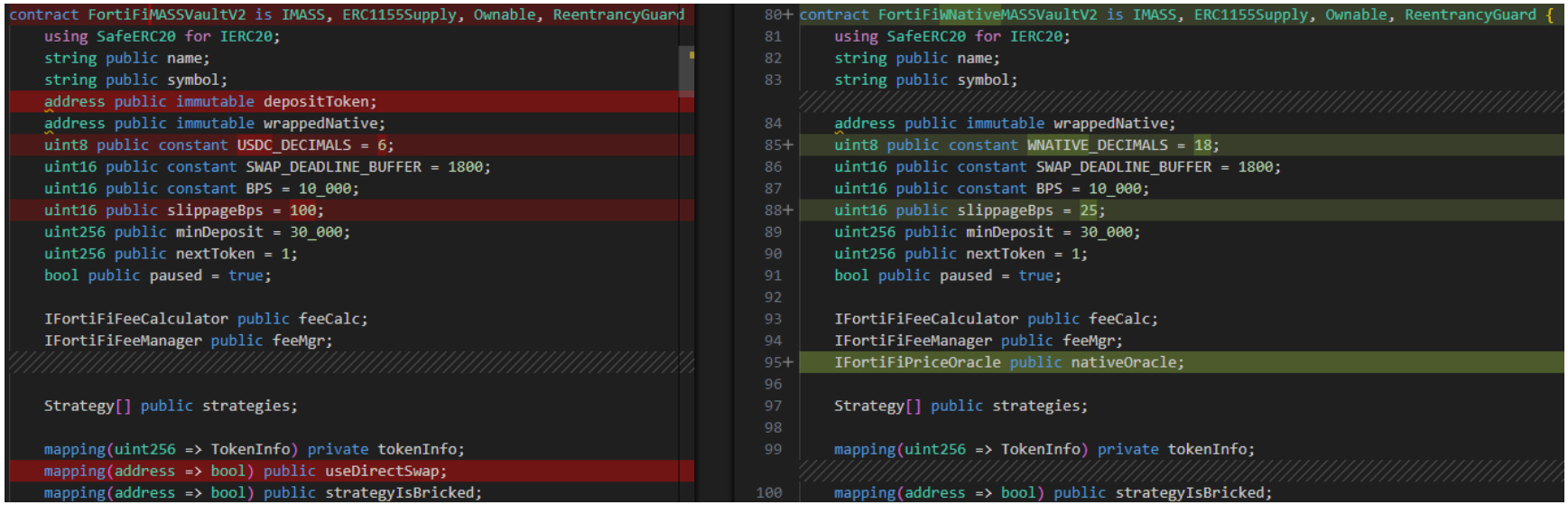
```
function setSlippage(uint16 _amount) external onlyOwner {
    if (_amount < 10 || _amount > 500) revert InvalidSlippage();
    slippageBps = _amount;
}
```

FortiFiWNativeMASSVaultV2

This contract was marked V2 as it required the removal of some functionality, namely the ability to utilize SAMS vaults as an underlying strategy as well as the setBpsForStrategies function. This was required to add the strategyIsBricked logic because the contract size was too large to compile. You can the V1 contract here: FortiFiWNativeMASSVault.

NOTE Native MASS vaults were not part of the original audit. These vaults were forked from the original MASS vaults to allow for utilizing wrapped native tokens as the deposit token instead of USDC. This section of the report will highlight the differences between FortiFiMASSVaultV2 and FortiFiWNativeMASSVaultV2.

Variable Declaration Changes

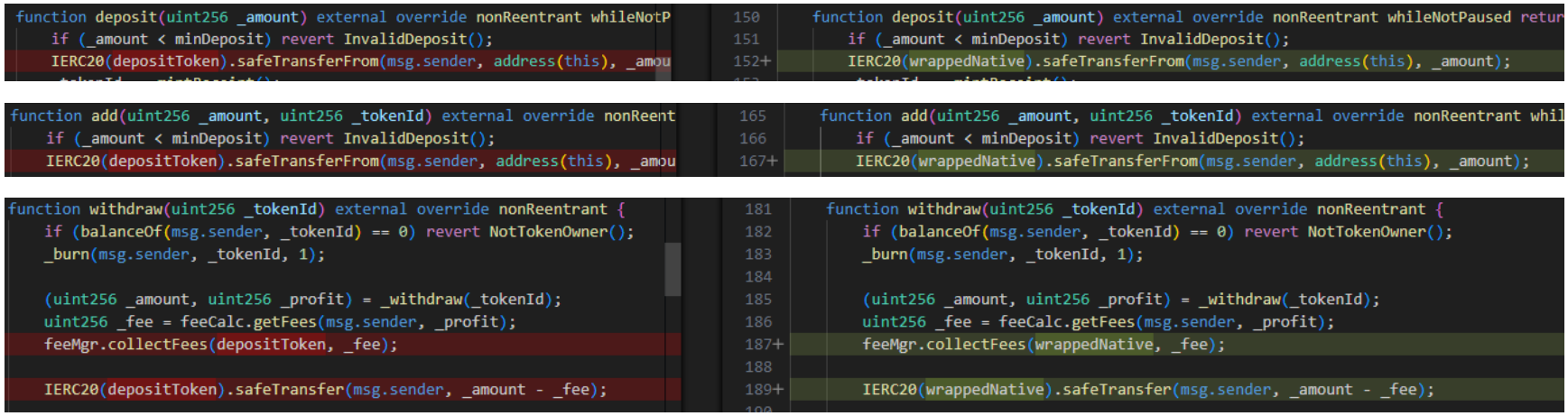


```
contract FortiFiMASSVaultV2 is IMASS, ERC1155Supply, Ownable, ReentrancyGuard
    using SafeERC20 for IERC20;
    string public name;
    string public symbol;
    address public immutable depositToken;
    address public immutable wrappedNative;
    uint8 public constant USDC_DECIMALS = 6;
    uint16 public constant SWAP_DEADLINE_BUFFER = 1800;
    uint16 public constant BPS = 10_000;
    uint16 public slippageBps = 100;
    uint256 public minDeposit = 30_000;
    uint256 public nextToken = 1;
    bool public paused = true;

    IFortiFiFeeCalculator public feeCalc;
    IFortiFiFeeManager public feeMgr;


    Strategy[] public strategies;

    mapping(uint256 => TokenInfo) private tokenInfo;
    mapping(address => bool) public useDirectSwap;
    mapping(address => bool) public strategyIsBricked;
```

```
contract FortiFiWNativeMASSVaultV2 is IMASS, ERC1155Supply, Ownable, ReentrancyGuard {
    using SafeERC20 for IERC20;
    string public name;
    string public symbol;


    address public immutable wrappedNative;
    uint8 public constant WNATIVE_DECIMALS = 18;
    uint16 public constant SWAP_DEADLINE_BUFFER = 1800;
    uint16 public constant BPS = 10_000;
    uint16 public slippageBps = 25;
    uint256 public minDeposit = 30_000;
    uint256 public nextToken = 1;
    bool public paused = true;

    IFortiFiFeeCalculator public feeCalc;
    IFortiFiFeeManager public feeMgr;
    IFortiFiPriceOracle public nativeOracle;

    Strategy[] public strategies;

    mapping(uint256 => TokenInfo) private tokenInfo;


    mapping(address => bool) public strategyIsBricked;
```

a) No more depositToken
b) WNATIVE_DECIMALS used instead of USDC_DECIMALS.
c) Addition of global nativeOracle for native token price used in swaps
d) useDirectSwap mapping no longer needed since all swaps are direct

Deposit, Add, and Withdraw functions updated to use wrappedNative instead of depositToken



```
function deposit(uint256 _amount) external override nonReentrant whileNotP
    if (_amount < minDeposit) revert InvalidDeposit();
    IERC20(depositToken).safeTransferFrom(msg.sender, address(this), _amou
```

```
150    function deposit(uint256 _amount) external override nonReentrant whileNotPaused retur
151        if (_amount < minDeposit) revert InvalidDeposit();
152+       IERC20(wrappedNative).safeTransferFrom(msg.sender, address(this), _amount);
```

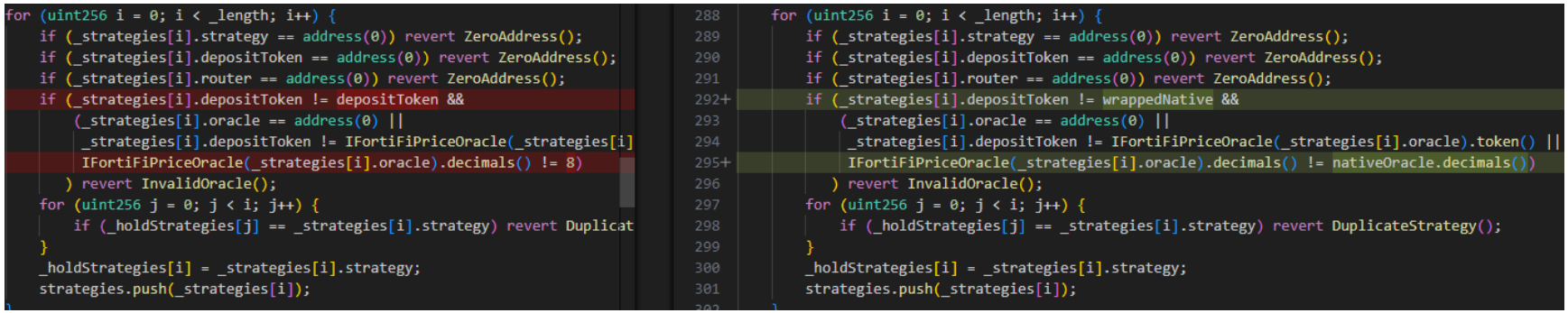
```
function add(uint256 _amount, uint256 _tokenId) external override nonReent
    if (_amount < minDeposit) revert InvalidDeposit();
    IERC20(depositToken).safeTransferFrom(msg.sender, address(this), _amou
```

```
165    function add(uint256 _amount, uint256 _tokenId) external override nonReentrant whil
166        if (_amount < minDeposit) revert InvalidDeposit();
167+       IERC20(wrappedNative).safeTransferFrom(msg.sender, address(this), _amount);
```

```
function withdraw(uint256 _tokenId) external override nonReentrant {
    if (balanceOf(msg.sender, _tokenId) == 0) revert NotTokenOwner();
    _burn(msg.sender, _tokenId, 1);

    (uint256 _amount, uint256 _profit) = _withdraw(_tokenId);
    uint256 _fee = feeCalc.getFees(msg.sender, _profit);
    feeMgr.collectFees(depositToken, _fee);

    IERC20(depositToken).safeTransfer(msg.sender, _amount - _fee);
```

```
181    function withdraw(uint256 _tokenId) external override nonReentrant {
182        if (balanceOf(msg.sender, _tokenId) == 0) revert NotTokenOwner();
183        _burn(msg.sender, _tokenId, 1);
184
185        (uint256 _amount, uint256 _profit) = _withdraw(_tokenId);
186        uint256 _fee = feeCalc.getFees(msg.sender, _profit);
187+       feeMgr.collectFees(wrappedNative, _fee);
188
189+       IERC20(wrappedNative).safeTransfer(msg.sender, _amount - _fee);
```

Required Oracle Configuration is checked slightly differently



```
for (uint256 i = 0; i < _length; i++) {
    if (_strategies[i].strategy == address(0)) revert ZeroAddress();
    if (_strategies[i].depositToken == address(0)) revert ZeroAddress();
    if (_strategies[i].router == address(0)) revert ZeroAddress();
    if (_strategies[i].depositToken != depositToken &&
        (_strategies[i].oracle == address(0) ||
         _strategies[i].depositToken != IFortiFiPriceOracle(_strategies[i]
         IFortiFiPriceOracle(_strategies[i].oracle).decimals() != 8)
       ) revert InvalidOracle();
    for (uint256 j = 0; j < i; j++) {
        if (_holdStrategies[j] == _strategies[i].strategy) revert Duplicat
    }
    _holdStrategies[i] = _strategies[i].strategy;
    strategies.push(_strategies[i]);
```

```
288    for (uint256 i = 0; i < _length; i++) {
289        if (_strategies[i].strategy == address(0)) revert ZeroAddress();
290        if (_strategies[i].depositToken == address(0)) revert ZeroAddress();
291        if (_strategies[i].router == address(0)) revert ZeroAddress();
292+       if (_strategies[i].depositToken != wrappedNative &&
293            (_strategies[i].oracle == address(0) ||
294             _strategies[i].depositToken != IFortiFiPriceOracle(_strategies[i].oracle).token() ||
295+            IFortiFiPriceOracle(_strategies[i].oracle).decimals() != nativeOracle.decimals())
296           ) revert InvalidOracle();
297        for (uint256 j = 0; j < i; j++) {
298            if (_holdStrategies[j] == _strategies[i].strategy) revert DuplicateStrategy();
299        }
300        _holdStrategies[i] = _strategies[i].strategy;
301        strategies.push(_strategies[i]);
```

Swap logic is modified for calculating using the nativeOracle

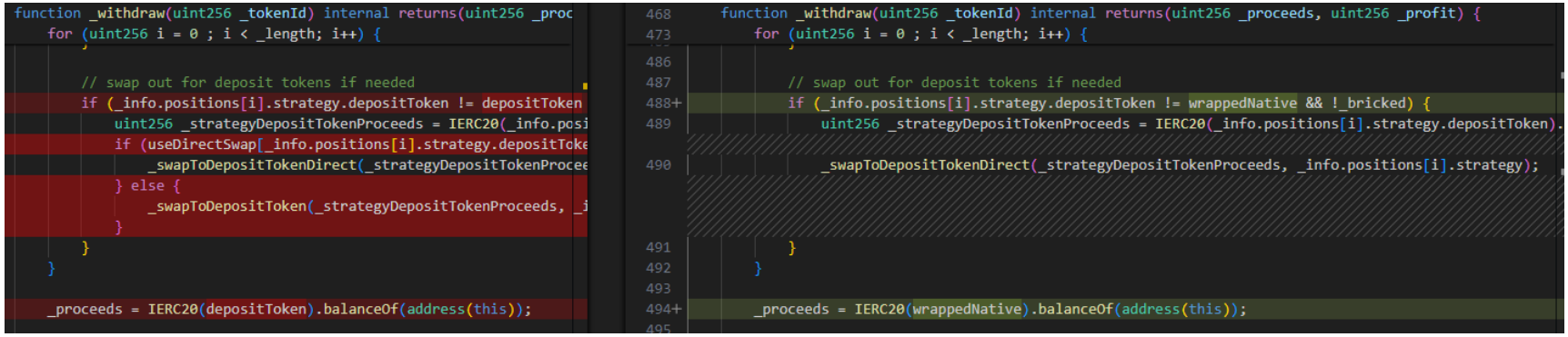
```
function _swapFromDepositTokenDirect(uint256 _amount, Strategy memory _strat) internal returns(uint256) {
    address _strategyDepositToken = _strat.depositToken;
    address[] memory _route = new address[](2);
    IRouter _router = IRouter(_strat.router);
    IFortiFiPriceOracle _oracle = IFortiFiPriceOracle(_strat.oracle);

    _route[0] = wrappedNative;
    _route[1] = _strategyDepositToken;

    uint256 _latestPriceNative = nativeOracle.getPrice();
    uint256 _latestPriceTokenB = _oracle.getPrice();
    uint256 _swapAmount = _amount * _latestPriceNative * 10**18 / _latestPriceTokenB / 10**18 / 10**(WNATIVE_DECIMALS - _strat.decimals);
```

```
function _swapToDepositTokenDirect(uint256 _amount, Strategy memory _strat) internal {
    address _strategyDepositToken = _strat.depositToken;
    address[] memory _route = new address[](2);
    IRouter _router = IRouter(_strat.router);
    IFortiFiPriceOracle _oracle = IFortiFiPriceOracle(_strat.oracle);

    _route[0] = _strategyDepositToken;
    _route[1] = wrappedNative;

    uint256 _latestPriceNative = nativeOracle.getPrice();
    uint256 _latestPriceTokenB = _oracle.getPrice();
    uint256 _swapAmount = _amount * _latestPriceTokenB / 10**18 / _latestPriceNative * 10**18 * 10**(WNATIVE_DECIMALS - _strat.decimals);
```

Removal of direct swap conditionals

```
function _deposit(uint256 _amount, uint256 _tokenId, bool _isAdd) i
    for (uint256 i = 0; i < _length; i++) {
        // split deposit and swap if necessary
        if (i == (_length - 1)) {
            if (depositToken != _strategy.depositToken) {
                if (useDirectSwap[_strategy.depositToken]) {
                    _depositAmount = _swapFromDepositTokenDirect(_rem
                } else {
                    _depositAmount = _swapFromDepositToken(_remainder
                }
            } else {
                _depositAmount = _remainder;
            }
        } else {
            uint256 _split = _amount * _strategy.bps / BPS;
            if (depositToken != _strategy.depositToken) {
                if (useDirectSwap[_strategy.depositToken]) {
                    _depositAmount = _swapFromDepositTokenDirect(_spl
                } else {
                    _depositAmount = _swapFromDepositToken(_split, _s
                }
            } else {
```

```
411     function _deposit(uint256 _amount, uint256 _tokenId, bool _isAdd) internal {
416         for (uint256 i = 0; i < _length; i++) {
428             // split deposit and swap if necessary
429             if (i == (_length - 1)) {
430+                if (wrappedNative != _strategy.depositToken) {
431                     _depositAmount = _swapFromDepositTokenDirect(_remainder, _strategy);
432                 } else {
433                     _depositAmount = _remainder;
434                 }
435             } else {
436                 uint256 _split = _amount * _strategy.bps / BPS;
437+                if (wrappedNative != _strategy.depositToken) {
438                     _depositAmount = _swapFromDepositTokenDirect(_split, _strategy);
439                 } else {
```



```
function _withdraw(uint256 _tokenId) internal returns(uint256 _proc
    for (uint256 i = 0 ; i < _length; i++) {

        // swap out for deposit tokens if needed
        if (_info.positions[i].strategy.depositToken != depositToken
            uint256 _strategyDepositTokenProceeds = IERC20(_info.posi
            if (useDirectSwap[_info.positions[i].strategy.depositToke
                _swapToDepositTokenDirect(_strategyDepositTokenProcee
            } else {
                _swapToDepositToken(_strategyDepositTokenProceeds, _i
            }
        }
    }

    _proceeds = IERC20(depositToken).balanceOf(address(this));
```

```
468     function _withdraw(uint256 _tokenId) internal returns(uint256 _proceeds, uint256 _profit) {
473         for (uint256 i = 0 ; i < _length; i++) {
486
487             // swap out for deposit tokens if needed
488+            if (_info.positions[i].strategy.depositToken != wrappedNative && !_bricked) {
489                 uint256 _strategyDepositTokenProceeds = IERC20(_info.positions[i].strategy.depositToken).
490                 _swapToDepositTokenDirect(_strategyDepositTokenProceeds, _info.positions[i].strategy);
491             }
492         }
493
494+        _proceeds = IERC20(wrappedNative).balanceOf(address(this));
```

Refund logic modified

```
/// @notice Internal function to refund left over tokens from deposit
function _refund(TokenInfo memory _info) internal {
    // Refund left over deposit tokens, if any
    uint256 _depositTokenBalance = IERC20(depositToken).balanceOf(adc
    if (_depositTokenBalance > 0) {
        _info.deposit -= _depositTokenBalance;
        IERC20(depositToken).safeTransfer(msg.sender, _depositTokenBa
    }

    // Refund left over wrapped native tokens, if any
    uint256 _wrappedNativeTokenBalance = IERC20(wrappedNative).balanc
    if (_wrappedNativeTokenBalance > 0) {
        IERC20(wrappedNative).safeTransfer(msg.sender, _wrappedNative
    }
```

```
503     /// @notice Internal function to refund left over tokens from deposit/add/rebalance transactions
504     function _refund(TokenInfo memory _info) internal {
505         // Refund left over deposit tokens, if any
506+        uint256 _depositTokenBalance = IERC20(wrappedNative).balanceOf(address(this));
507         if (_depositTokenBalance > 0) {
508             _info.deposit -= _depositTokenBalance;
509+            IERC20(wrappedNative).safeTransfer(msg.sender, _depositTokenBalance);
510         }
511
```

FortiFiStrategy & FortiFiVectorStrategy

1. Addition of ‘strategyIsBricked’ boolean

   Problem: It was determined that users may be unable to withdraw their deposits if one of the underlying strategies became broken (i.e. when Platypus was exploited, users who deposited into certain pools on Yield Yak were unable to withdraw from the pool).

   Solution: Allow the strategy to become ‘bricked’, in which case the protocol can call withdrawBricked on the fortress and send the user the receipt tokens for that specific underlying strategy instead of withdrawing and returning deposit tokens.

   Implementation: Similar to the logic added to SAMS and MASS vaults.

```
bool public strategyIsBricked;
```

```
function depositToFortress(uint256 _amount, address _user, uint256 _tokenId) ext
    if (_amount == 0) revert InvalidDeposit();
    if (!isFortiFiVault[msg.sender]) revert InvalidCaller();
    if (strategyIsBricked) revert StrategyBricked();
```

```
function withdrawFromFortress(uint256 _amount, address _user, uint256 _tokenId) external virtual {
    if (_amount == 0) revert InvalidWithdrawal();
    if (vaultToTokenToFortress[msg.sender][_tokenId] == address(0)) revert NoFortress();

    // burn receipt tokens and withdraw from Fortress
    _burn(msg.sender, _amount);

    if(strategyIsBricked) {
        IFortress(vaultToTokenToFortress[msg.sender][_tokenId]).withdrawBricked(_user);

        emit WithdrawBrickedFromFortress(msg.sender, _user, address(_strat), _tokenId);
    } else {
        IFortress(vaultToTokenToFortress[msg.sender][_tokenId]).withdraw(_user, extraTokens);
```

2. Addition of extraTokens array

Problem: It was found that some yield-producing strategies arbitrarily add new reward tokens to their protocols. If unhandled by the contract these reward tokens could become locked in fortresses and difficult or impossible to retrieve.

Solution: Store a modifiable array of token contract addresses that can be iterated through to determine if additional tokens were received upon withdrawal. Call a FortiFiFeeCalculator and FortiFiFeeManager in order to take fees from these tokens before sending the remainder to the user.

Implementation:

a) Add required variables

```
IFortiFiFeeCalculator public feeCalc;
IFortiFiFeeManager public feeMgr;

address[] public extraTokens;
```

b) Handle extra tokens

```
function withdrawFromFortress(uint256 _amount, address _user, uint256 _tokenId) external virtual {
    } else {
        _dToken.safeTransfer(msg.sender, _depositTokenReceived);

        // handle fees on extra reward tokens
        uint256 _length = extraTokens.length;
        if (_length > 0) {
            for(uint256 i = 0; i < _length; i++) {
                IERC20 _token = IERC20(extraTokens[i]);
                uint256 _tokenBalance = _token.balanceOf(address(this));
                if (_tokenBalance > 0) {
                    uint256 _fee = feeCalc.getFees(_user, _tokenBalance);
                    _token.approve(address(feeMgr), _fee);
                    feeMgr.collectFees(extraTokens[i], _fee);
                    _token.safeTransfer(_user, _tokenBalance - _fee);
                }
            }
        }
```

c) Add setters for extra tokens, fee manager, and fee calculator

```solidity
/// @notice Set extra reward tokens for strategy
function setExtraTokens(address[] calldata _tokens) external onlyOwner {
    extraTokens = _tokens;
    emit ExtraTokensSet(_tokens);
}

/// @notice Set strategy as bricked
function setStrategyBricked(bool _isBricked) external onlyOwner {
    strategyIsBricked = _isBricked;
    emit SetStrategyAsBricked(_isBricked);
}

/// @notice Function to set new FortiFiFeeManager contract
function setFeeManager(address _contract) external onlyOwner {
    if (_contract == address(0)) revert ZeroAddress();
    feeMgr = IFortiFiFeeManager(_contract);
    emit FeeManagerSet(_contract);
}

/// @notice Function to set new FortiFiFeeCalculator contract
function setFeeCalculator(address _contract) external onlyOwner {
    if (_contract == address(0)) revert ZeroAddress();
    feeCalc = IFortiFiFeeCalculator(_contract);
    emit FeeCalculatorSet(_contract);
}
```

FortiFiFortress & FortiFiVectorFortress

1. Addition of 'strategyIsBricked' boolean

   Problem: It was determined that users may be unable to withdraw their deposits if one of the underlying strategies became broken (i.e. when Platypus was exploited, users who deposited into certain pools on Yield Yak were unable to withdraw from the pool).

   Solution: Add withdrawBricked function that sends the user the receipt tokens for that specific underlying strategy instead of withdrawing and returning deposit tokens.

   Implementation:

```
function withdrawBricked(address _user) external virtual onlyOwner {
    uint256 _balance = _strat.balanceOf(address(this));
    if (_balance == 0) revert InvalidWithdrawal();

    // ensure no strategy receipt tokens remain
    _balance = _strat.balanceOf(address(this));
    if (_balance > 0) {
        IERC20(address(_strat)).safeTransfer(_user, _balance);
    }

    emit WithdrawalMade(_user);
}
```

2. Support for extraTokens

   Problem: It was found that some yield-producing strategies arbitrarily add new reward tokens to their protocols. If unhandled by the contract these reward tokens could become locked in fortresses and difficult or impossible to retrieve.

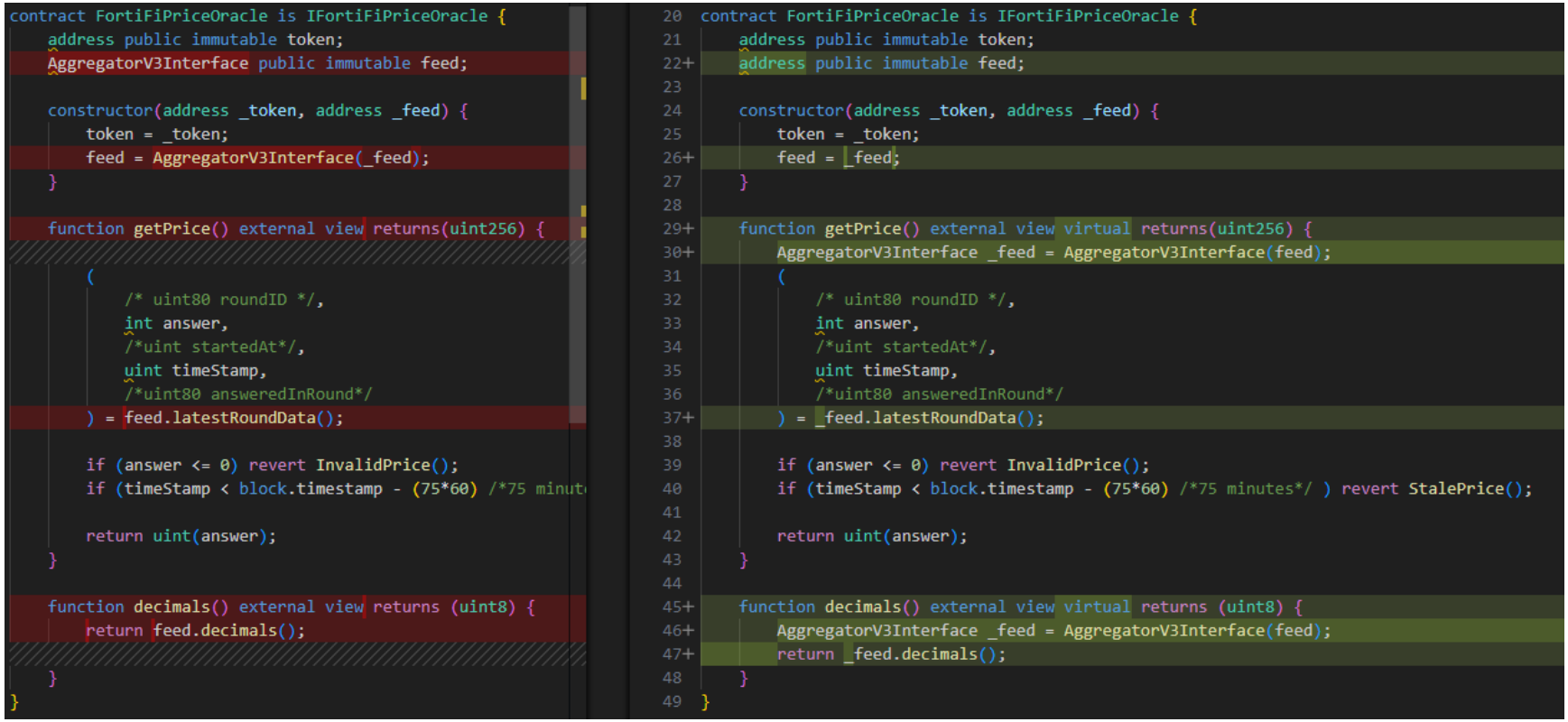
Solution: Pass an array of token contract addresses in the withdraw function that can be iterated through to determine if additional tokens were received upon withdrawal. Call a FortiFiFeeCalculator and FortiFiFeeManager in order to take fees from these tokens before sending the remainder to the user.

Implementation:

```
function withdraw(address _user, address[] memory _extraTokens) external virtual onlyOwner {

    // transfer received deposit tokens and refund left over tokens, if any
    _dToken.safeTransfer(msg.sender, _dToken.balanceOf(address(this)));

    // transfer extra reward tokens
    uint256 _length = _extraTokens.length;
    if (_length > 0) {
        for(uint256 i = 0; i < _length; i++) {
            IERC20 _token = IERC20(_extraTokens[i]);
            uint256 _tokenBalance = _token.balanceOf(address(this));
            if (_tokenBalance > 0) {
                _token.safeTransfer(msg.sender, _tokenBalance);
            }
        }
    }
```

FortiFiPriceOracle

Changes were made to make the oracle contract more flexible



```
contract FortiFiPriceOracle is IFortiFiPriceOracle {
    address public immutable token;
    AggregatorV3Interface public immutable feed;

    constructor(address _token, address _feed) {
        token = _token;
        feed = AggregatorV3Interface(_feed);
    }

    function getPrice() external view returns(uint256) {

        (
            /* uint80 roundID */,
            int answer,
            /*uint startedAt*/,
            uint timeStamp,
            /*uint80 answeredInRound*/
        ) = feed.latestRoundData();

        if (answer <= 0) revert InvalidPrice();
        if (timeStamp < block.timestamp - (75*60) /*75 minut

        return uint(answer);
    }

    function decimals() external view returns (uint8) {
        return feed.decimals();

    }
}
```

```
contract FortiFiPriceOracle is IFortiFiPriceOracle {
    address public immutable token;
    address public immutable feed;

    constructor(address _token, address _feed) {
        token = _token;
        feed = _feed;
    }

    function getPrice() external view virtual returns(uint256) {
        AggregatorV3Interface _feed = AggregatorV3Interface(feed);
        (
            /* uint80 roundID */,
            int answer,
            /*uint startedAt*/,
            uint timeStamp,
            /*uint80 answeredInRound*/
        ) = _feed.latestRoundData();

        if (answer <= 0) revert InvalidPrice();
        if (timeStamp < block.timestamp - (75*60) /*75 minutes*/ ) revert StalePrice();

        return uint(answer);
    }

    function decimals() external view virtual returns (uint8) {
        AggregatorV3Interface _feed = AggregatorV3Interface(feed);
        return _feed.decimals();
    }
}
```

FortiFiRouter

FortiFiRouter contracts were created to act as an interface between MASS vaults and concentrated liquidity dexes. The allow for the vaults to pass in swap parameters for a V1 swapExactTokensForTokens call and use those parameters to execute more advanced swaps. This is a new concept that was not audited by Blaize.

New Contracts

This table lists all new contracts and notes about areas that should be examined

Contract	Notes
FortiFiPriceOracleL2 FortiFiDIAPriceOracle FortiFiDIAPriceOracleL2 FortiFiMockPriceOracle	These oracle contracts are only slight variations of the original FortiFiPriceOracle contract. L2 oracles implement Chainlink's sequencer uptime feed FortiFiMockPriceOracle simply returns a static value. This is to allow for "routers" that don't actually swap to still flow through the vault logic. An example of this is the FortiFiGGAvaxRouter (see below).
FortiFiGGAvaxRouter	This router can only be used to either: 1. Unwrap WAVAX and use ggAVAX's deposit function in order to mint ggAVAX 2. Use ggAVAX's redeem function to burn ggAVAX, receive AVAX, and wrap the AVAX in order to return WAVAX. Potential concern is this could prevent withdrawals if there is not enough AVAX in gogopool's reserve to allow for the redeeming of ggAVAX. Currently there seems to be sufficient liquidity that this isn't a concern.

FortiFiLBRouter FortiFiLBRouter2	FortiFiLBRouter is only meant to be used for WAVAX/[asset] pairs. FortiFiLBRouter2 can be used for any LB pair. There is a failsafe mechanism whereby the owner of the contract can change the version and binSteps in order to have the swap utilize Trader Joe V1 pools if they exist. This is in case the LB pair no longer has sufficient liquidity. There is some concern around what happens when Trader Joe updates to new versions, such as the v2.2 that is soon to be released. TJ has stated via their Discord support that the v2.2 upgrade should not impact contracts that have hard-coded v2.1 as the interface for swaps is the same. As a precaution, V2_2 and V3 were added to the Versions struct so that they may potentially be used in the future if possible.
FortiFiUniV3Router FortiFiUniV3MultiHopRouter	These routers utilize Uniswap V3 exactInput calls. The MultiHop router is to be used for assets that do not have a direct pair, and will execute the swap by using wrapped native assets as an intermediate step.
FortiFiNativeStrategy FortiFiNativeFortress	This strategy/fortress pair is designed to be used in FortiFiWNativeMASSVaults. The strategy will accept wrapped native tokens as a deposit, and the fortress will unwrap/wrap as needed for deposits and withdrawals.

FortiFiGLPStrategy FortiFiGLPStrategyArb FortiFiGLPFortress FortiFiGLPFortressArb	These strategy/fortress pairs accept USDC deposits and mint/burn GLP directly through the GLP Reward Router. The only difference in the Arb contracts is that they have Arb GLP contracts hard-coded, whereas the non-Arb contracts have Avalanche GLP contracts hard-coded. This could be simplified to use constructor args instead in a future iteration. Reward router, staked GLP, and slippage can be updated at the strategy level. At the fortress level, the strategy owner must call 'updateForFortress' functions and pass in all the fortress addresses. While this shouldn't ever happen, it will be a cumbersome undertaking should it be needed and this logic has room for improvement. One area of potential concern is the the GLP price used when calculating how much to mint is retrieved from the GLP Manager, on-chain. Normally fetching prices on-chain is susceptible to manipulations, but that doesn't seem to be the case with GLP. GLP price is very stable, and there is a cooldown period between minting and redeeming that makes sandwich attacks unlikely.

FortiFiWombatStrategy FortiFiWombatFortress	This strategy/fortress pair is used to deposit into a Wombat single-sided LP, and then stake those LP tokens with Yield Yak. Because the LP is single-sided, it has been noted that deposits typically yield an amount of receipt tokens equal to the amount of the deposit in wei. A small slippage amount is applied to deposits. Withdrawals currently use no slippage, with users able to redeem at least the amount of depositTokens that they initially deposited (minus deposit slippage), but withdrawal slippage can be set separately.