BakerFi Recursive Staking

February 2024

Made with ❤ by the following Creed authors: Gonçalo Sá, Dominik Muhs

Table of Contents

Executive Summary

This report presents the results of our engagement with BakerFi to review their smart contracts. The review was conducted over 3 weeks, from **February 26, 2024 to March 15, 2024** by Gonçalo Sá and Dominik Muhs. A total of six person-weeks were spent.

Throughout this engagement, we identified 15 findings of varying severity. Among these are four major-severity findings. No critical findings have been identified. The major issues concern areas such as the integration with external oracles, aspects of the deployment process that may compromise security, and the presence of hardcoded values within the contract code.

A follow-up review of the code base has been performed from **May 6, 2024 to May 10, 2024** by Dominik Muhs. The mitigations for the previously identified issues have been reviewed and marked as fixed where applicable.

Scope and Objectives

Our review focused on the commit hash `7b3234a631039a3c7d633ffcef2e61666bf4324c`. Together with the BakerFi team, we identified the following priorities for our review:

- Review the security of oracle integrations and user flows potentially resulting in the loss of funds.
- Ensure that the system is implemented consistently with the intended functionality, and without unintended edge cases.
- Identify known vulnerabilities particular to smart contract systems, as outlined in our Smart Contract Security Field Guide, and the ones outlined in the EEA EthTrust Security Levels Specification.

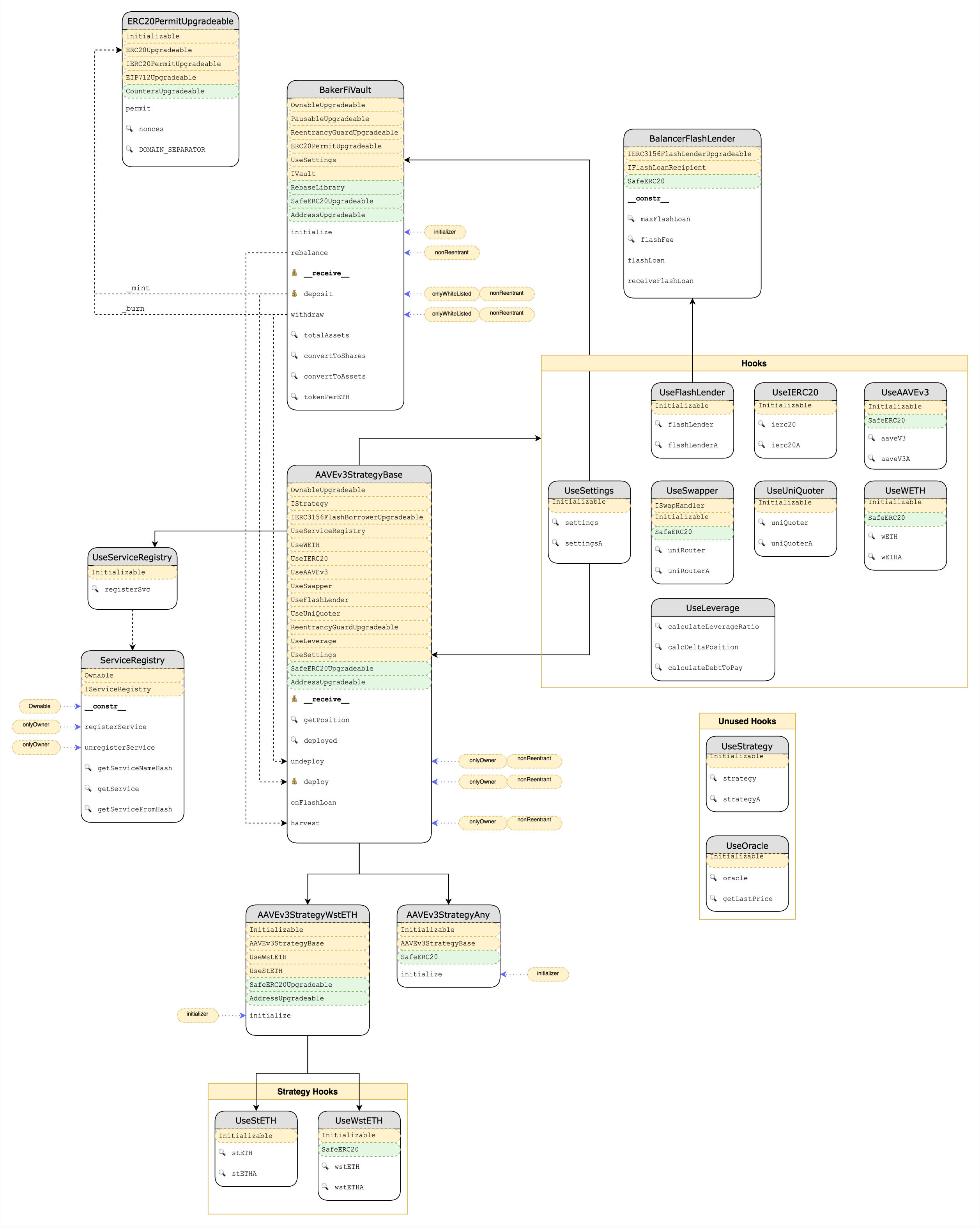
The follow-up review of mitigations has been performed on the following commit hash: `55becd7da43c9e769f8b73ff9e9bd9f344c77388`. On May 8, additional changes from a different branch were introduced to the scope under the following commit hash: `8e547937b0d1d9bde8099d684c60e0240a309b44`. Inside this change set, only code relevant to the mitigations has been reviewed.

Audit Artifacts

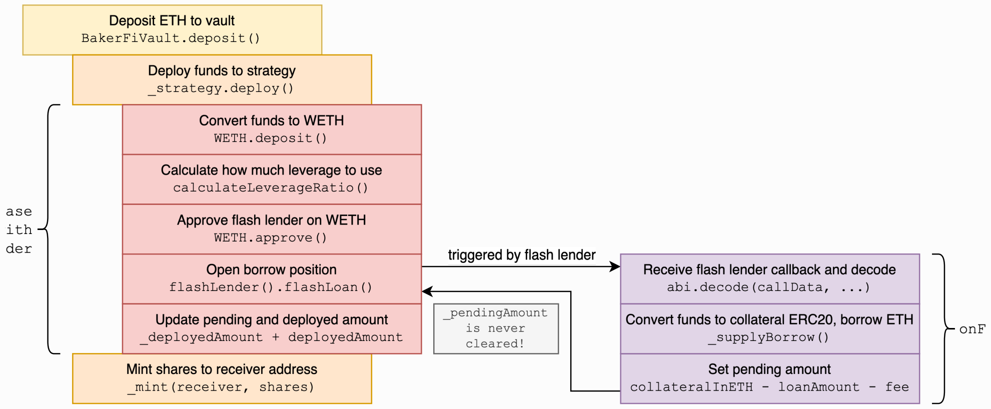
System Architecture

During our engagement with BakerFi's development team, we discussed the system's architecture and identified key areas of concern. Notably, the one-to-one relationship between the Vault and Strategy components, potential for architectural optimization, and the complexity introduced by the "hook" system were highlighted as areas of future work. We recommended the simplification of the overall code base to improve the system maintenance characteristics and reduce potential points of failure.

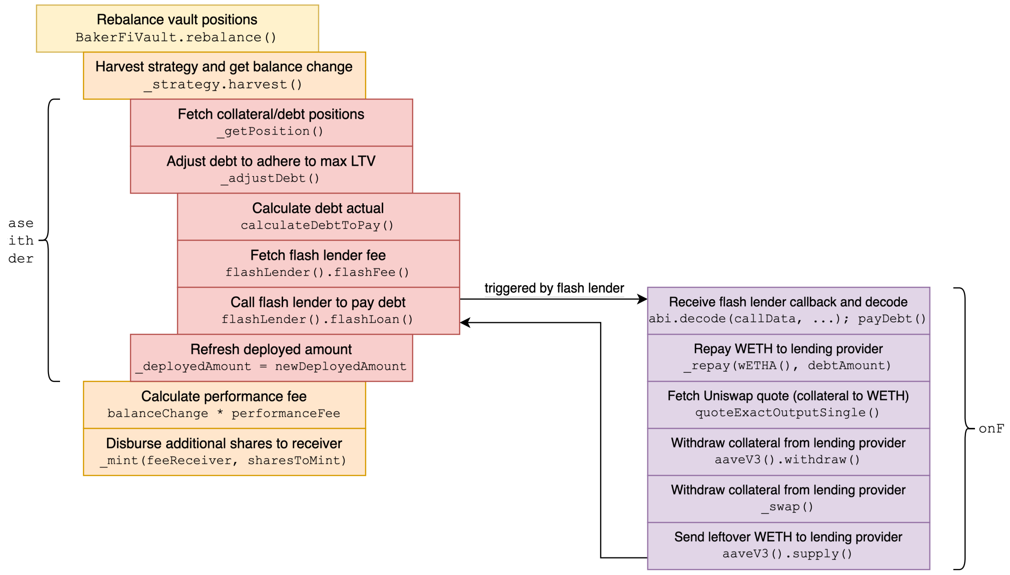
During the mitigations review, it has been found that the system architecture has remained largely the same, specifically the relationship between Vault and Strategy contracts, as well as the redundant complexity of the "hook" system. For this reason, the architecture diagram below has remained largely the same.


ERC20PermitUpgradeable
Initializable
ERC20Upgradeable
IERC20PermitUpgradeable
EIP712Upgradeable
CountersUpgradeable
permit
nonces
DOMAIN_SEPARATOR
_mint
_burn
BakerFiVault
OwnableUpgradeable
PausableUpgradeable
ReentrancyGuardUpgradeable
ERC20PermitUpgradeable
UseSettings
IVault
RebaseLibrary
SafeERC20Upgradeable
AddressUpgradeable
initialize
rebalance
__receive__
deposit
withdraw
totalAssets
convertToShares
convertToAssets
tokenPerETH
initializer
nonReentrant
onlyWhiteListed
nonReentrant
onlyWhiteListed
nonReentrant
BalancerFlashLender
IERC3156FlashLenderUpgradeable
IFlashLoanRecipient
SafeERC20
__constr__
maxFlashLoan
flashFee
flashLoan
receiveFlashLoan
Hooks
UseFlashLender
Initializable
flashLender
flashLenderA
UseIERC20
Initializable
ierc20
ierc20A
UseAAVEv3
Initializable
SafeERC20
aaveV3
aaveV3A
UseSettings
Initializable
settings
settingsA
UseSwapper
ISwapHandler
Initializable
SafeERC20
uniRouter
uniRouterA
UseUniQuoter
Initializable
uniQuoter
uniQuoterA
UseWETH
Initializable
SafeERC20
wETH
wETHA
UseLeverage
calculateLeverageRatio
calcDeltaPosition
calculateDebtToPay
AAVEv3StrategyBase
OwnableUpgradeable
IStrategy
IERC3156FlashBorrowerUpgradeable
UseServiceRegistry
UseWETH
UseIERC20
UseAAVEv3
UseSwapper
UseFlashLender
UseUniQuoter
ReentrancyGuardUpgradeable
UseLeverage
UseSettings
SafeERC20Upgradeable
AddressUpgradeable
__receive__
getPosition
deployed
undeploy
deploy
onFlashLoan
harvest
onlyOwner
nonReentrant
onlyOwner
nonReentrant
onlyOwner
nonReentrant
UseServiceRegistry
Initializable
registerSvc
ServiceRegistry
Ownable
IServiceRegistry
__constr__
registerService
unregisterService
getServiceNameHash
getService
getServiceFromHash
Ownable
onlyOwner
onlyOwner
Unused Hooks
UseStrategy
Initializable
strategy
strategyA
UseOracle
Initializable
oracle
getLastPrice
AAVEv3StrategyWstETH
Initializable
AAVEv3StrategyBase
UseWstETH
UseStETH
SafeERC20Upgradeable
AddressUpgradeable
initialize
initializer
AAVEv3StrategyAny
Initializable
AAVEv3StrategyBase
SafeERC20
initialize
initializer
Strategy Hooks
UseStETH
Initializable
stETH
stETHA
UseWstETH
Initializable
SafeERC20
wstETH
wstETHA

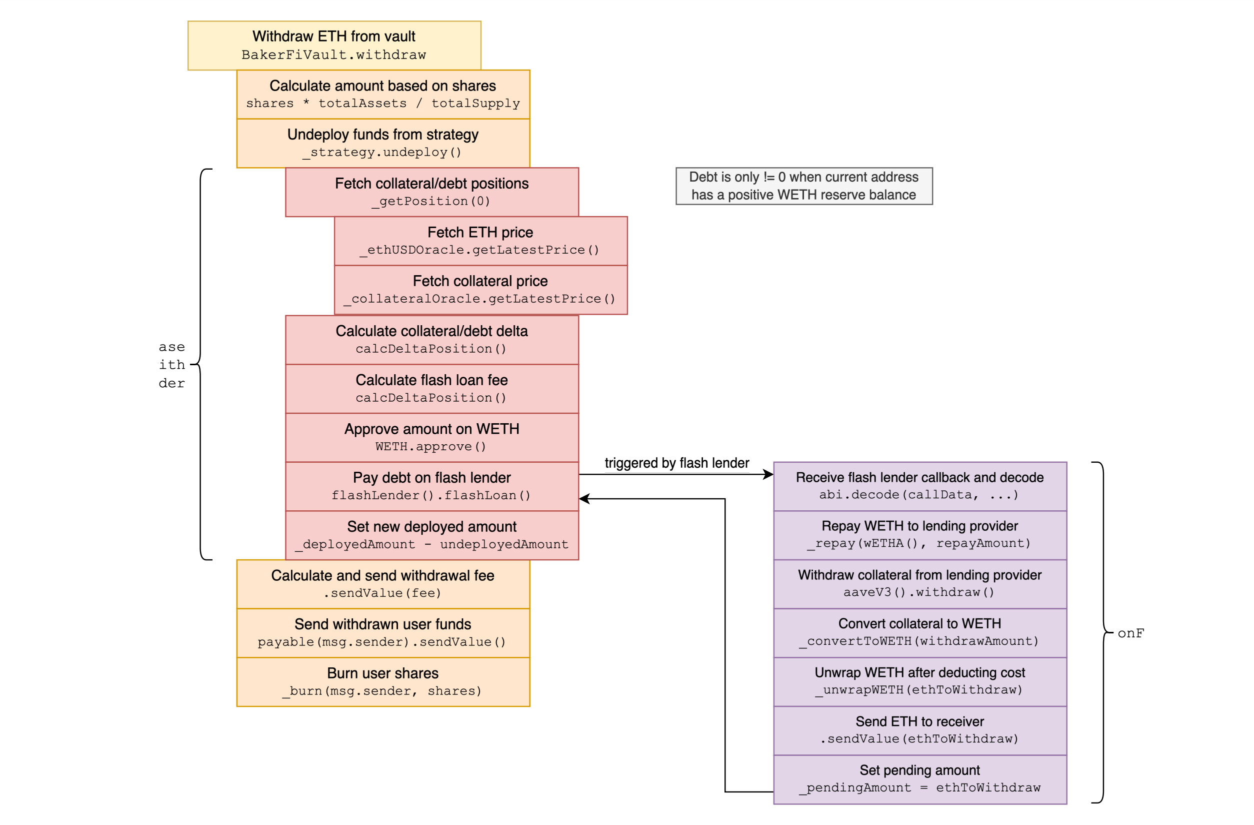
Deposit Flow


Deposit ETH to vault
BakerFiVault.deposit()
Deploy funds to strategy
_strategy.deploy()
Convert funds to WETH
WETH.deposit()
Calculate how much leverage to use
calculateLeverageRatio()
Approve flash lender on WETH
WETH.approve()
Open borrow position
flashLender().flashLoan()
Update pending and deployed amount
_deployedAmount + deployedAmount
Mint shares to receiver address
_mint(receiver, shares)
ase
ith
der
triggered by flash lender
_pendingAmount
is never
cleared!
Receive flash lender callback and decode
abi.decode(callData, ...)
Convert funds to collateral ERC20, borrow ETH
_supplyBorrow()
Set pending amount
collateralInETH - loanAmount - fee
onF

Rebalance Flow


Rebalance vault positions
BakerFiVault.rebalance()
Harvest strategy and get balance change
_strategy.harvest()
Fetch collateral/debt positions
_getPosition()
Adjust debt to adhere to max LTV
_adjustDebt()
Calculate debt actual
calculateDebtToPay()
Fetch flash lender fee
flashLender().flashFee()
Call flash lender to pay debt
flashLender().flashLoan()
Refresh deployed amount
_deployedAmount = newDeployedAmount
Calculate performance fee
balanceChange * performanceFee
Disburse additional shares to receiver
_mint(feeReceiver, sharesToMint)
ase
ith
der
triggered by flash lender
Receive flash lender callback and decode
abi.decode(callData, ...); payDebt()
Repay WETH to lending provider
_repay(wETHA(), debtAmount)
Fetch Uniswap quote (collateral to WETH)
quoteExactOutputSingle()
Withdraw collateral from lending provider
aaveV3().withdraw()
Withdraw collateral from lending provider
_swap()
Send leftover WETH to lending provider
aaveV3().supply()
onF

Withdrawal Flow


Withdraw ETH from vault
BakerFiVault.withdraw
Calculate amount based on shares
shares * totalAssets / totalSupply
Undeploy funds from strategy
_strategy.undeploy()
Fetch collateral/debt positions
_getPosition(0)
Debt is only != 0 when current address has a positive WETH reserve balance
Fetch ETH price
_ethUSDOracle.getLatestPrice()
Fetch collateral price
_collateralOracle.getLatestPrice()
ase
ith
der
Calculate collateral/debt delta
calcDeltaPosition()
Calculate flash loan fee
calcDeltaPosition()
Approve amount on WETH
WETH.approve()
Pay debt on flash lender
flashLender().flashLoan()
triggered by flash lender
Set new deployed amount
_deployedAmount - undeployedAmount
Calculate and send withdrawal fee
.sendValue(fee)
Send withdrawn user funds
payable(msg.sender).sendValue()
Burn user shares
_burn(msg.sender, shares)
Receive flash lender callback and decode
abi.decode(callData, ...)
Repay WETH to lending provider
_repay(wETHA(), repayAmount)
Withdraw collateral from lending provider
aaveV3().withdraw()
Convert collateral to WETH
_convertToWETH(withdrawAmount)
onF
Unwrap WETH after deducting cost
_unwrapWETH(ethToWithdraw)
Send ETH to receiver
.sendValue(ethToWithdraw)
Set pending amount
_pendingAmount = ethToWithdraw

Findings

Major Calculations On Stale Price Data During Withdrawal

Fixed

> This issue is considered fixed in the following commit hash: `fd6ece3473fc27a1a8e4f6521b62f9c5181cd46d`. The maximum price age is now dynamically fetched from the settings.

During the withdrawal process, a user interacts with the `BakerFiVault.withdraw` function, which internally calls `AAVEv3StrategyBase._undeploy`. This function is responsible for retrieving the current collateral and debt values to calculate the amount of assets to be withdrawn and the corresponding debt to be repaid.

contracts/core/strategies/AAVEv3StrategyBase.sol

```
(uint256 totalCollateralBaseInEth, uint256 totalDebtBaseInEth) = _getPosition(0);
// When the position is in liquidation state revert the transaction
require(
    totalCollateralBaseInEth > totalDebtBaseInEth,
    "No Collateral margin to scale"
);
```

During this flow, `priceMaxAge` is used with a value of zero in the `_getPosition` call, which effectively skips the timeliness check of the oracle's answer. This allows the withdrawal process to proceed with potentially outdated price data.

contracts/core/strategies/AAVEv3StrategyBase.sol

```
require(priceMaxAge == 0 ||
    (priceMaxAge > 0  && (ethPrice.lastUpdate  >= (block.timestamp - priceMaxAge…
    (priceMaxAge > 0  && (collateralPrice.lastUpdate >= (block.timestamp - price…
, "Oracle Price is outdated");
```

Consequently, the computation of the delta positions in `_undeploy` may be based on stale information, leading to an incorrect number of shares being burned during the withdrawal operation.

contracts/core/BakerFiVault.sol

```
amount = _strategy.undeploy(withdrawAmount);
uint256 fee = 0;
// Withdraw ETh to Receiver and pay withdrawal Fees
if (settings().getWithdrawalFee() != 0  && settings().getFeeReceiver() != addres…
    fee = amount * settings().getWithdrawalFee() /PERCENTAGE_PRECISION;
    payable(msg.sender).sendValue(amount - fee);
    payable(settings().getFeeReceiver()).sendValue(fee);
} else {
    payable(msg.sender).sendValue(amount);
}
_burn(msg.sender, shares);
```

Recommendation

To mitigate this risk, we recommend considering the implementation of a system similar to AAVEv3's `PriceOracleSentinel`, which assesses the health of the price oracle and either allows or denies certain actions based on this evaluation. Alternatively, explicit checks to validate the timeliness and accuracy of the oracle data should be implemented. The mitigration should ensure that operations relying on price data are executed with current and accurate information and prevent potentially stale price data from influencing the internal accounting.

Major Chainlink - Deprecated Integration And Lax Validation

Fixed

> This issue has been fixed in the following commit hash: `fd6ece3473fc27a1a8e4f6521b62f9c5181cd46d`. The Chainlink-integrating contracts now use `latestRoundData` and validate the answer and timestamps accordingly.

The project's codebase integrates oracles from both Pyth and Chainlink to retrieve price data. However, the Chainlink oracle integrations lack necessary validation checks on the oracle responses. This greatly increases the risk of incorporating stale or incorrect data into critical calculations, such as those determining debt levels and collateral values.

Specifically, the current implementations in `ETHOracle`, `WstETHToETHOracleETH`, and `CbETHToETHOracle` use the `latestAnswer` function to fetch price data:

- `ETHOracle.getLatestPrice`:

contracts/oracles/EthOracle.sol

```
price.price = uint256(_ethPriceFeed.latestAnswer()*1e10);
price.lastUpdate = _ethPriceFeed.latestTimestamp();
```

- `WstETHToETHOracleETH`:

contracts/oracles/WstETHToETHOracleETH.sol

```
uint256 stETHToETH = uint256(_stETHToETHPriceFeed.latestAnswer());
price.price = wstETHToStETH * stETHToETH / _PRECISION;
price.lastUpdate = _stETHToETHPriceFeed.latestTimestamp();
```

- `CbETHToETHOracle`:

contracts/oracles/cbETHToETHOracle.sol

```
function getLatestPrice() external override view returns (IOracle.Price memory p…
    price.price = uint256(_stCbETHToETHPriceFeed.latestAnswer());
    price.lastUpdate = _stCbETHToETHPriceFeed.latestTimestamp();
}
```

This approach is problematic for several reasons. The `latestAnswer` method is deprecated and offers limited information. It also returns limited data, making it harder to validate the return value's freshness and accuracy.

Recommendation

To address these issues, we recommend switching from using `latestAnswer` to `latestRoundData`. The `latestRoundData` function returns more values that enable further validation checks:

- It allows for checking if the returned raw price is greater than `0`, ensuring that the price data is meaningful and not indicative of an error or placeholder value.
- It enables validation that the `updatedAt` timestamp is not `0`, which would indicate that the round data is incomplete or uninitialized.
- The method provides both `startedAt` and `updatedAt` timestamps, eliminating the need for additional external calls, reducing complexity and saving gas.

Major Deployment Vulnerable To Frontrunning

Fixed

> This issue has been fixed in the following commit hash: `0f97236cc8e1abe3b14fb59d91eb9c02337e1182`. The deployments are now done through the ethers contract factory, which allows passing the encoded initialize call in the same step.

The deployment process for system components involves deploying the implementation contract code and subsequently calling the `initialize` method through a proxy. This two-step approach, while common in upgradeable smart contract systems, introduces a vulnerability where an attacker might front-run the deployment of individual components.

scripts/common.ts

```
const strategy = await AAVEv3Strategy.deploy();
await strategy.waitForDeployment();
```

Specifically, an attacker could execute a transaction that calls `initialize` on the newly deployed contract before the legitimate initializer has the chance to do so. This could allow the attacker to set up malicious configurations, including the owner, service registry, collateral tokens, and oracles, with potentially subtle differences that may be hard to detect.

scripts/common.ts

```
await strategy.initialize(
  owner,
  serviceRegistry,
  ethers.keccak256(Buffer.from(collateral)),
  ethers.keccak256(Buffer.from(oracle)),
  swapFreeTier,
  emodeCategory
);
```

The deployment script's current process does not include checks to validate the integrity of deployed contracts against unauthorized initializations.

Recommendation

To mitigate the risk of such front-running attacks and unauthorized initializations, it's recommended to either:

- **Deploy Upgradeable Components via a Smart Contract Factory:** Use a smart contract factory that handles both deployment and initialization within a single atomic transaction. This approach ensures that there is no window of opportunity for an attacker to intervene between the deployment and initialization phases.
- **Introduce a Guardian Role:** Implement a guardian role in the system components that restricts the calling of the `initialize` function to authorized addresses only. This guardian role would act as a safeguard, making sure that only designated addresses (e.g., those belonging to authorized deployers) can initialize the contract.

Updating the deployment script to include validation checks post-deployment can give additional guarantees that the contract's state matches the intended configuration.

Major Balancer - Hardcoded Fee

Fixed

> This issue has been fixed in the following commit hash: `0c5249ba3d398b0d5b7551caae1d8e9bf1870b87`. Now, the `BalancerFlashLender` contract dynamically fetches the flash fee percentage.

The `BalancerFlashLender` contract currently has the Balancer flash loan fees hardcoded to zero. This poses a risk because Balancer's governance can vote to enable flash loan fees in the future.

contracts/core/flashloan/BalancerFlashLender.sol

```
function flashFee(address, uint256) external pure override returns (uint256) {
    return 0;
}
```

If Balancer governance decides to implement flash loan fees, the hardcoded value will become inaccurate. This discrepancy will affect interactions within the `AAVEv3StrategyBase.deploy` method, leading to an incorrect amount of WETH being approved for the flash loan. As a result, the flash loan operation could fail, causing the deposit process to revert until an appropriate upgrade is done.

contracts/core/strategies/AAVEv3StrategyBase.sol

```
uint256 fee = flashLender().flashFee(wETHA(), loanAmount);
//§uint256 allowance = wETH().allowance(address(this), flashLenderA());
require(wETH().approve(flashLenderA(), loanAmount + fee));
require(
    flashLender().flashLoan(
        IERC3156FlashBorrowerUpgradeable(this),
        wETHA(),
        loanAmount,
        abi.encode(msg.value, msg.sender, FlashLoanAction.SUPPLY_BOORROW)
    ),
    "Failed to run Flash Loan"
);
```

Recommendation

We recommend dynamically fetching the flash loan fee percentage rather than relying on a hardcoded value. Specifically, the Balancer vault contract provides a way to obtain the current flash loan fee percentage through the protocol fee collector. The fee can be retrieved with the following call: `_balancerVault.getProtocolFeesCollector().getFlashLoanFeePercentage()`.

This approach allows the strategy to automatically adjust to any future changes in Balancer's flash loan fee policy. Generally, it is a best practice to avoid hardcoding values that depend on third-party services, as these can change over time.

Medium Lax Pyth Oracle Validation In `getPriceUnsafe`

Fixed

> This issue has been addressed in the following commit hash: `f17fca72cfc20f428ac49c8b62fcdf055536462d`. A missing function call to `getSafeLatestPrice` was identified after and fixed in the following commit hash: `8e547937b0d1d9bde8099d684c60e0240a309b44`. These changes have been pushed to a separate, out-of-scope development branch.

In the `PythOracle` contract, the `getPriceUnsafe` method is used to fetch price information. This method does not automatically revert when the given price is stale, which could lead to it returning a price that is arbitrarily far in the past. The `price.publishTime` returned by this function is not validated within the function itself to ensure that the price feed data is timely. Instead, the responsibility for this validation is passed onto the caller. This approach increases the risk of developer error and could potentially lead to duplicated code across the system where the function is used.

contracts/oracles/PythOracle.sol

```
function _getPriceInternal() private view returns (IOracle.Price memory outPrice…
    PythStructs.Price memory price = _pyth.getPriceUnsafe(_priceID);

    if (price.expo >= 0) {
        outPrice.price = uint64(price.price) * uint256(10**(_precisison+uint32(p…
    } else {
        outPrice.price = uint64(price.price) * uint256(10**(_precisison-uint32(-…
    }
    outPrice.lastUpdate = price.publishTime;
}
```

Recommendation

We recommend implementing an internal validation mechanism within the `getPriceUnsafe` function itself. The function should check the `price.publishTime` against a predefined threshold to ensure that the price data is not too old. If the data is found to be stale, the function should revert with an appropriate error message. This will centralize the validation logic, reducing the risk of developer error and avoiding duplicated validation code in other parts of the system.

Medium

License Violation Regarding BoringCrypto Dependency

Fixed

> This issue has been fixed in the following commit hash: `a8713f787bf6162c4b6d36742e479a4d421c035e`.

The `BoringRebase` library within the project is taken from the BoringCrypto smart contract suite, raising concerns regarding adherence to copyright and licensing terms. The comparison reveals that the code implemented in the project is a substantial, partial copy of the original BoringCrypto library, available under an MIT license. This license permits commercial usage but obligates users to include the original copyright and license notification in any significant reproductions or distributions of the licensed material.

```
contracts/libraries/BoringRebase.sol

library RebaseLibrary {

    /// @notice Calculates the base value in relationship to `elastic` and `tota…
    function toBase(
        Rebase memory total,
        uint256 elastic,
        bool roundUp
    ) internal pure returns (uint256 base) {
        if (total.elastic == 0) {
            base = elastic;
        } else {
            base = (elastic * total.base) / total.elastic;
            if (roundUp && (base * total.elastic) / total.base < elastic) {
                base++;
            }
        }
    }

    /// @notice Calculates the elastic value in relationship to `base` and `tota…
    function toElastic(
        Rebase memory total,
        uint256 base,
        bool roundUp
    ) internal pure returns (uint256 elastic) {
        if (total.base == 0) {
            elastic = base;
        } else {
            elastic = (base * total.elastic) / total.base;
            if (roundUp && (elastic * total.base) / total.elastic < base) {
                elastic++;
            }
        }
    }
}
```

The original `BoringRebase` library code: https://github.com/boringcrypto/BoringSolidity/blob/78f4817d9c0d95fe9c45cd42e307ccd22cf5f4fc/contracts/libraries/BoringRebase.sol

Given the importance of complying with copyright laws and respecting the work of original authors, we recommend ensuring that the project's usage of the `BoringRebase` library includes proper attribution to the original source and adheres to the MIT license conditions.

Furthermore, we recommend consulting with a legal expert familiar with intellectual property rights in software to obtain precise guidance on maintaining compliance and avoiding potential licensing infringements in the future.

Medium

Funds Received By BakerFiVault Will Be Lost

Fixed

> This issue has been fixed in `fd6ece3473fc27a1a8e4f6521b62f9c5181cd46d` by restricting the `receive` function to only the vault's authorized strategy contract.

Funds sent directly to the `BakerFiVault` contract are at risk of being irrevocably stuck because the contract lacks the necessary logic to account for or withdraw these funds:

contracts/core/BakerFiVault.sol

```
/**
 * @dev Fallback function to receive Ether.
 *
 * This function is marked as external and payable. It is automatically called
 * when Ether is sent to the contract, such as during a regular transfer or as p…
 * of a self-destruct operation.
 *
 * Emits no events and allows the contract to accept Ether.
 */
receive() external payable {}
```

Additionally, the existing comment regarding the `receive` function's behavior in the event of a `selfdestruct` call is misleading. In reality, when a contract is destroyed via `selfdestruct`, the ETH balance is transferred to the specified receiver address without executing any code, including the `receive` function.

Recommendation

We recommend implementing safeguards against the direct transfer of ETH to the contract. This could be achieved by removing the `receive` function or by introducing an authorized emergency transfer function to allow the contract owner to retrieve and return the funds to their origin.

Medium Uninitialized Implementation

Fixed

> This issue has been fixed in the following commit hash: `0f97236cc8e1abe3b14fb59d91eb9c02337e1182`.

The `BakerFiVault` contract allows anyone to initialize its implementation as it does not properly disable its initializers. It is missing a constructor altogether:

contracts/core/BakerFiVault.sol

```
function initialize(
    address initialOwner,
    ServiceRegistry registry,
    IStrategy strategy
) public initializer {
    __ERC20Permit_init(_NAME);
    __ERC20_init(_NAME, _SYMBOL);
    _initUseSettings(registry);
    require(initialOwner != address(0), "Invalid Owner Address");
    _transferOwnership(initialOwner);
    _registry = registry;
    _strategy = strategy;
}
```

Recommendation

We recommend the addition of a constructor within the `BakerFiVault` with a call to `_disableInitializers`. This change will ensure that the contract is properly initialized upon deployment.

Minor Checks-Effects-Interactions Violation

Fixed

> This issue has been resolved in the following commit hash: `a8713f787bf6162c4b6d36742e479a4d421c035e`.

In the `BakerFiVault` contract, particularly within the `withdraw` method, there's a violation of the checks-effects-interactions pattern:

contracts/core/BakerFiVault.sol

```
if (settings().getWithdrawalFee() != 0  && settings().getFeeReceiver() != addres…
    fee = amount * settings().getWithdrawalFee() /PERCENTAGE_PRECISION;
    payable(msg.sender).sendValue(amount - fee);
    payable(settings().getFeeReceiver()).sendValue(fee);
} else {
    payable(msg.sender).sendValue(amount);
}
_burn(msg.sender, shares);
emit Withdraw(msg.sender, amount - fee, shares);
```

Recommendation

We recommend refactoring the `withdraw` method to strictly follow the checks-effects-interactions pattern. This means ensuring that all necessary validation checks and state changes are performed before any external calls are made. Going into the external call, the vault's shares should already have been updated, preventing the callee having access to partial state changes.

Minor Duplicate Ownable Initialization

Fixed

> This issue has been resolved in the following commit hash: `427de1bcc50d9e164f8555e62bd3a994f120ef3a`. Redundant `Ownable` initializations have been removed in favor of explicit `_transferOwnership` calls.

The `ServiceRegistry` smart contract is designed to be deployed with its constructor directly initializing the `Ownable` dependency. Within the `Ownable` contract, the constructor assigns the contract deployer as the owner by calling `_transferOwnership(msg.sender)`. However, insite the constructor body, a subsequent call to `_transferOwnership` is made, passing a parameter to set the correct owner, effectively repeating the ownership assignment.

contracts/core/ServiceRegistry.sol

```
constructor(address ownerToSet) Ownable()
{
    require(ownerToSet != address(0), "Invalid Owner Address");
    _transferOwnership(ownerToSet);
}
```

A similar issue affects the `BKR` token contract:

contracts/core/governance/BKR.sol

```
constructor(address initialOwner)
    ERC20(_NAME, _SYMBOL)
    Ownable()
    ERC20Permit(_NAME)
    ERC20Votes()
{
```

Recommendation

We recommend removing the redundant `Ownable` initialization in the constructor header. This not only avoids unnecessary code but also reduces the deployment cost by eliminating redundant operations.

Minor

Unused Code

Fixed

> The unused code, along with other redundancies, has been removed as of the latest commit hash.

The codebase includes instances of unused hooks and imports, as well as state variables that are initialized but never used within the contract's logic. These redundancies not only clutter the code but also potentially complicate maintenance and readability.

One example is the unused import in `BalancerFlashLender.sol`:

contracts/core/flashloan/BalancerFlashLender.sol

```
import {UseStrategy} from "../../core/hooks/UseStrategy.sol";
```

Furthermore, the `registry` parameter in `BakerFiVault.sol` is required by the `initialize` function to call `_initUseSettings` but is subsequently assigned at the contract level without being used:

contracts/core/BakerFiVault.sol

```
_transferOwnership(initialOwner);
_registry = registry;
_strategy = strategy;
```

Recommendation

To enhance the codebase's readability and maintenance efficiency, we recommend the removal of all unused hooks, imports, and variables. Specifically:

- Identify and eliminate any unused imports across the project, such as the one found in `BalancerFlashLender.sol`. Tools such as linters or static code analysis can assist in this process by highlighting unused code areas.
- Review and remove unused contract hooks and other code that does not contribute to the system's functionality. This includes any hooks declared but not integrated into the contract's logic.
- Refactor any parameters or variables that are initialized but not used in the contract's business logic. For example, the `registry` parameter in `BakerFiVault.sol` should be removed if it has no purpose beyond the initial setup.

Minor Inconsistent Use Of `UseServiceRegistry`

Fixed

> This issue has been fixed in the following commit hash: `0ffb57f652062f2c1a1641b44adb55008b589440`. The `UseServiceRegistry` hook has been removed and each system component now accesses its own `registry` state variable directly.

The current implementation inconsistently uses the `ServiceRegistry` contract across different parts of the system. The `UseServiceRegistry` "hook" is designed to initialize and manage access to the `ServiceRegistry` contract, serving as an abstraction layer that isolates and controls interactions with it.

The `AAVEv3StrategyBase` contract demonstrates proper use of the `UseServiceRegistry` hook:

contracts/core/strategies/AAVEv3StrategyBase.sol

```
abstract contract AAVEv3StrategyBase is
    OwnableUpgradeable,
    IStrategy,
    IERC3156FlashBorrowerUpgradeable,
    UseServiceRegistry,
    UseWETH,
    UseIERC20,
    UseAAVEv3,
    UseSwapper,
    UseFlashLender,
    UseUniQuoter,
    ReentrancyGuardUpgradeable,
    UseLeverage,
    UseSettings
{
```

However, the `BakerFiVault` contract bypasses the `UseServiceRegistry` abstraction, directly interacting with the `ServiceRegistry` through its own `_registry` state variable. This approach deviates from the intended design pattern, leading to inconsistencies in how services are accessed and managed across the system.

contracts/core/BakerFiVault.sol

```
/**
 * @dev The ServiceRegistry contract used for managing service-related dependenc…
 *
 * This private state variable holds the reference to the ServiceRegistry contra…
 * that is utilized within the current contract for managing various service dep…
 */
ServiceRegistry private _registry;
```

Recommendation

All contracts within the system that require access to the `ServiceRegistry` should consistently utilize the `UseServiceRegistry` hook.

Furthermore, we recommend analyzing the current usage and access patterns to the service registry across the system. Based on this, the `UseServiceRegistry` hook should be refined to provide an explicit interface for interacting with the `ServiceRegistry`. This might involve defining specific methods within the hook that abstract common interactions, rather than directly exposing the entire `ServiceRegistry` contract.

Minor Missing Separation Of Concerns In Library Code

Fixed

> This issue has been fixed in the following commit hash: `8e9140a76d29f959a40367a269794e4183827143`

To improve clarity and maintain the principle of separation of concerns, it's crucial to distinguish between the `BoringCrypto` library's production code and its test contract. Currently, the test contract is embedded within the library file, which could lead to confusion.

contracts/libraries/BoringRebase.sol

```
contract TestRebaseLibrary {

    using RebaseLibrary for Rebase;

    function toBase(
        Rebase memory total,
        uint256 elastic,
        bool roundUp) public pure returns (uint) {
        return total.toBase(
            elastic,
            roundUp
        );
    }

    function toElastic(
        Rebase memory total,
        uint256 base,
        bool roundUp
    ) public pure returns (uint256 elastic) {
        return total.toElastic(
            base,
            roundUp
        );
    }
}
```

Recommendation

We recommend extracting the test contract from the library files and relocating it to a separate directory explicitly designated for test code.

Minor

Direct Vault Ownership Transfer

Fixed

> This issue has been fixed in the following commit hash: `4aa7d92c5cf5b4ee056430805b7f8a23b359e66f`.

The `BakerFiVault` contract currently implements a single-step ownership transfer mechanism. This approach lacks an intermediate verification step, which increases the risk of accidental or unauthorized ownership changes due to developer error or malicious actions.

contracts/core/BakerFiVault.sol

```
contract BakerFiVault is
    OwnableUpgradeable,
    PausableUpgradeable,
    ReentrancyGuardUpgradeable,
    ERC20PermitUpgradeable,
    UseSettings,
    IVault
{
```

Recommendation

We recommend modifying the ownership transfer process to a two-step procedure. This enhancement would require the new owner address to actively confirm its readiness to assume ownership before the transfer is finalized. Furthermore, we discourage the use of a single private key in favor of a multi-signature wallet to mitigate the impact of a party losing access to their private key or having it compromised.

None Redundant ABIEncoderV2 Pragma

Fixed

> The redundant pragma has been removed and the code base has been upgraded to the latest Solidity compiler version.

The `BakerFiVault` contract includes the `ABIEncoderV2` pragma, which has become redundant for Solidity versions 0.8.0 and above:

contracts/core/BakerFiVault.sol

```
pragma experimental ABIEncoderV2;
```

More info: https://docs.soliditylang.org/en/latest/080-breaking-changes.html#silent-changes-of-the-semantics

Recommendation

We recommend removing the redundant pragma to clean up the code and adhere to best practices with Solidity version 0.8.0 and later. If there is a specific requirement to explicitly declare the use of ABI encoder version 2, the correct pragma directive `pragma abicoder v2;` should be used.

File Hashes

- 1b3c1d42c63d58bfa127de25ce85675d9357e3c6add2309819641f45769192c8: ./contracts/core/Settings.sol
- e5701da4f7eb57efb7b9421f0e80e62b76a3dc91ec12bfac5e6b2c67e4c8d077: ./contracts/core/strategies/AAVEv3StrategyAny.sol
- ada654e59b5122d5465a2b2190e25d071af469bcf228f79321d1f175378f8720: ./contracts/core/strategies/AAVEv3StrategyBase.sol
- 9f9ed199bacf147ba2da30e5cc1b6b7903824b82e8fd278d633fdb904e392330: ./contracts/core/strategies/AAVEv3StrategyWstETH.sol
- 8877c97089b791f0884bd2f1a6a7d21becd29d4da0450c4cbe639aed969c904f: ./contracts/core/flashloan/BalancerFlashLender.sol
- 272dc785ddd56bc5b19d8ad3743aad837653802e00498f2cbac12c988087cdee: ./contracts/core/ServiceRegistry.sol
- 8faf95479e9d857302a9be891d5b8b3d4476f1e018fe8ccd4fc7d6f01de82b8e: ./contracts/core/Constants.sol
- 2948b853ddfdaecce47a9166a8db3fd6eaff47ef72443a0ff21aa38f739766ba: ./contracts/core/hooks/UseSwapper.sol
- 3d53c401a3d71d4e6d8f0f133e8b1a8b17bf3d92dd70b4563321b7c9ea0b5bec: ./contracts/core/hooks/UseLeverage.sol
- aaa162decac64a3ec75cdc16f14cf900c381c4b5fc7cd603473d295d70108560: ./contracts/core/hooks/UseIERC20.sol
- b43ea959a18d5566bc86e2a7eefca24f963e7a25a4b34718eaea8056bb2c2954: ./contracts/core/hooks/UseServiceRegistry.sol
- 1cb234c86a93ea52f1179db6d4a7b3f8c9ca58568700882b6243c41f319b536d: ./contracts/core/hooks/UseOracle.sol
- 57dfa69a4f496d142fa9d58ad81f6d1df3650b5fdf704781c2184d0a489a5093: ./contracts/core/hooks/UseAAVEv3.sol
- 058e0451004c980d4ae4f2abb098290b07ec2cdc8b0f1d638d938e6fd84cad46: ./contracts/core/hooks/UseWstETH.sol
- fd34039e2c691f6be93d5dfd4c0eab2691994782a10344f2e287533aeb1d6a25: ./contracts/core/hooks/UseFlashLender.sol
- e3f0b8ded0a37acbea45f825ce5fe6178064e96908a146b4c68f0ee99dc1f58e: ./contracts/core/hooks/UseStETH.sol
- 1eea5694b34dd81d763e14e70c102e175efac741d17640868796334d2baaf32b: ./contracts/core/hooks/UseStrategy.sol
- f30ce659e1d8003484aeac671b5207f99e760d89ed8d64bdbd5392f26282acd9: ./contracts/core/hooks/UseSettings.sol

- 07ed475f0e3d3ad7e0c3167e092c5c60a92b5a8b8d0f873c37f64d5ec8d86aa5: ./contracts/core/hooks/UseWETH.sol
- f5480e35539b811cb473897b69b8670ab0259b60ab7340ce85ca487ee73dbaf0: ./contracts/core/hooks/UseUniQuoter.sol
- 35370d9650388bc8941cb0a45317b73581f27b103fc92c304d890068b6a4c436: ./contracts/core/BakerFiVault.sol
- d517c33d0b3d82d119a9c8633af64738274f75d2239578fcab4af432ff2dbaae: ./contracts/core/governance/BakerFiGovernor.sol
- f3ea9e15e683a90f279a43a844b2c7322e61488c3685d4eae01a22aec7cf83f2: ./contracts/core/governance/Timelock.sol
- 035843808f66ee1bc9d72957ee51b9c18ab348eb02acfc7d0848a3707245dd84: ./contracts/core/governance/BKR.sol
- 40b6b25e4c1297037e1316695a23798bc93a777c4b1eb3ab76a0258873afcc8c: ./contracts/proxy/BakerFiProxy.sol
- 489bdf705aed711c133451432a18b917ea66db5f3e585e7fde18e5fa7b202537: ./contracts/proxy/BakerFiProxyAdmin.sol
- e9180762a074a0199108d1b520100975629573a70b374999941c7c939c2a403e: ./contracts/oracles/WstETHToETHOracleETH.sol
- ad0ea0b20fc9fb3cb68e8f3394e7a61261f66e350746216254cc2fee2b8bc78c: ./contracts/oracles/WstETHToETHOracle.sol
- 5ee96738e2ac5dea126f233bdaf6eebfa577b6a20ae7d6fd8a42f651a3cd97f8: ./contracts/oracles/EthOracle.sol
- 369913e0639a24467fa187cff72a3b9e54cab1ed4853227819aad6159a3486b9: ./contracts/oracles/PythOracle.sol
- 046c672800e49de5154beee5b92fc1998a5ef7f53878aef1a7a8efaf90c2929b: ./contracts/oracles/cbETHToETHOracle.sol
- 9be8944fbb758e1a398685fa94686bd2bcc7f5dc286822bd979c7fba5232950f: ./contracts/libraries/tokens/WETH.sol
- a473ee697268580b3777e557804843e68c6298ad3fd6bbdc492778209b0e5d84: ./contracts/libraries/BoringRebase.sol

Disclaimer

Creed ("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 Creed's own 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 CD. Such hyperlinks are provided for your reference and convenience only, and are the exclusive responsibility of such web sites' owners. You agree that CD are not responsible for the content or operation of such Web sites, and that 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 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. 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 CD.