hexens × TOKEMAK

DEC.24

SECURITY REVIEW REPORT FOR TOKEMAK

CONTENTS

ABOUT HEXENS

Hexens is a cybersecurity company that strives to elevate the standards of security in Web 3.0, create a safer environment for users, and ensure mass Web 3.0 adoption.

Hexens has multiple top-notch auditing teams specialized in different fields of information security, showing extreme performance in the most challenging and technically complex tasks, including but not limited to: Infrastructure Audits, Zero Knowledge Proofs / Novel Cryptography, DeFi and NFTs. Hexens not only uses widely known methodologies and flows, but focuses on discovering and introducing new ones on a day-to-day basis.

In 2022, our team announced the closure of a $4.2 million seed round led by IOSG Ventures, the leading Web 3.0 venture capital. Other investors include Delta Blockchain Fund, Chapter One, Hash Capital, ImToken Ventures, Tenzor Capital, and angels from Polygon and other blockchain projects.

Since Hexens was founded in 2021, it has had an impressive track record and recognition in the industry: Mudit Gupta - CISO of Polygon Technology - the biggest EVM Ecosystem, joined the company advisory board after completing just a single cooperation iteration. Polygon Technology, 1inch, Lido, Hats Finance, Quickswap, Layerswap, 4K, RociFi, as well as dozens of DeFi protocols and bridges, have already become our customers and taken proactive measures towards protecting their assets.

EXECUTIVE SUMMARY

OVERVIEW

This audit covered updates to the smart contracts of Tokemak V2. Besides other changes, it introduced new functionality to the Autopilot- and SwapRouter contracts to allow users to set custom swap routes when redeeming Autopool assets.

Our security assessment was a full review of the code differences, spanning a total of 1 week.

During our audit, we have identified 1 critical severity vulnerability that could have allowed an attacker to steal assets from smart contract wallets or protocols if the redeem call could be sandwiched.

We have also identified several minor severity vulnerabilities and code optimisations.

Finally, all of our reported issues were fixed by the development team and consequently validated by us.

We can confidently say that the overall security and code quality have increased after completion of our audit.

SCOPE

The analyzed resources are located on:

https://github.com/Tokemak/v2-core/tree/b1c44a19c57a0d75414e19caac9b1901d94d2596

The issues described in this report were fixed in the following commits:

https://github.com/Tokemak/v2-core/tree/ceec54d22d0ecee322567d30b6123303738241af

AUDITING DETAILS

STARTED
16.12.2024

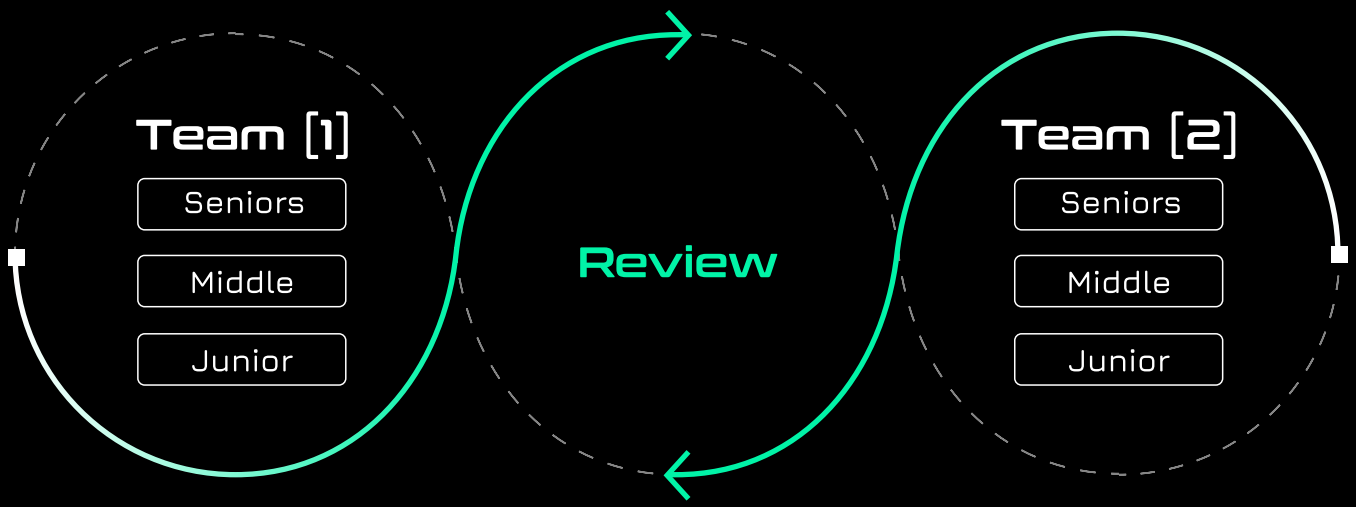
DELIVERED
26.12.2024

Review Led by

KASPER ZWIJSEN
Head of Audits | Hexens

HEXENS METHODOLOGY

Hexens methodology involves 2 teams, including multiple auditors of different seniority, with at least 5 security engineers. This unique cross-checking mechanism helps us provide the best quality in the market.



SEVERITY STRUCTURE

The vulnerability severity is calculated based on two components

- Impact of the vulnerability
- Probability of the vulnerability

Impact	Probability			
	rare	unlikely	likely	very likely
Low/Info	Low/Info	Low/Info	Medium	Medium
Medium	Low/Info	Medium	Medium	High
High	Medium	Medium	High	Critical
Critical	Medium	High	Critical	Critical

SEVERITY CHARACTERISTICS

Smart contract vulnerabilities can range in severity and impact, and it's important to understand their level of severity in order to prioritize their resolution. Here are the different types of severity levels of smart contract vulnerabilities:

Critical

Vulnerabilities with this level of severity can result in significant financial losses or reputational damage. They often allow an attacker to gain complete control of a contract, directly steal or freeze funds from the contract or users, or permanently block the functionality of a protocol. Examples include infinite mints and governance manipulation.

High

Vulnerabilities with this level of severity can result in some financial losses or reputational damage. They often allow an attacker to directly steal yield from the contract or users, or temporarily freeze funds. Examples include inadequate access control integer overflow/underflow, or logic bugs.

Medium

Vulnerabilities with this level of severity can result in some damage to the protocol or users, without profit for the attacker. They often allow an attacker to exploit a contract to cause harm, but the impact may be limited, such as temporarily blocking the functionality of the protocol. Examples include uninitialized storage pointers and failure to check external calls.

Low

Vulnerabilities with this level of severity may not result in financial losses or significant harm. They may, however, impact the usability or reliability of a contract. Examples include slippage and front-running, or minor logic bugs.

Informational

Vulnerabilities with this level of severity are regarding gas optimizations and code style. They often involve issues with documentation, incorrect usage of EIP standards, best practices for saving gas, or the overall design of a contract. Examples include not conforming to ERC20, or disagreement between documentation and code.

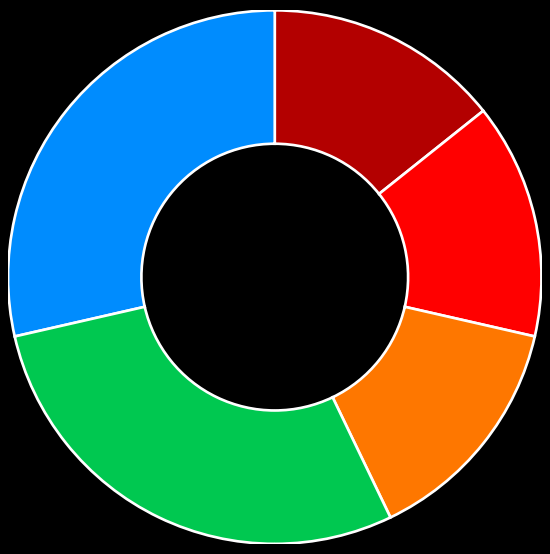
ISSUE SYMBOLIC CODES

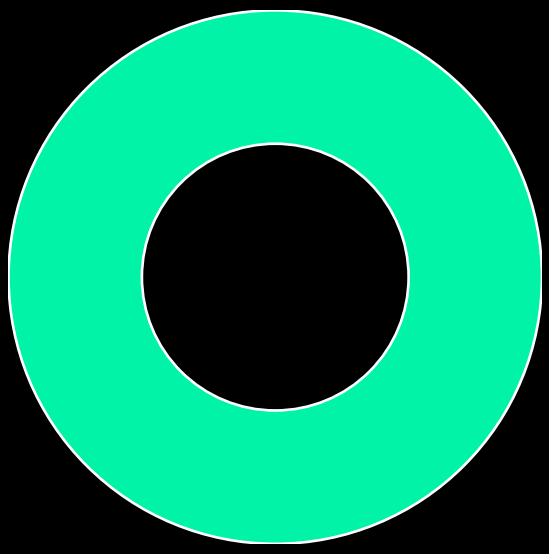
Every issue being identified and validated has its unique symbolic code assigned to the issue at the security research stage. Cause of the vulnerability reporting flow design, some of the rejected issues could be missing.

FINDINGS SUMMARY

Severity	Number of Findings
Critical	1
High	1
Medium	1
Low	2
Informational	2

Total: 7





WEAKNESSES

This section contains the list of discovered weaknesses.

TOKE4-6

TRANSIENT SWAP ROUTES CAN BE SET FOR OTHER SMART CONTRACT USERS TO STEAL TOKENS

SEVERITY: Critical

PATH:

src/vault/AutopilotRouter.sol:redeemWithRoutes

REMEDIATION:

The AutopilotRouter should have a re-entrancy lock in at least the case where custom user swap routes have been set.

STATUS: Fixed

DESCRIPTION:

The AutopilotRouter has introduced a new feature called the dynamic swap router, where a user is able to set their own swap path using transient storage.

In the function **redeemWithRoutes**, it first sets the swap path and then simply calls **redeem** on the vault. When the vault calls back to the **SwapRouterV2.swapForQuote**, it checks whether a swap path was set in the transient storage and it will use that one if it was set.

Because transient storage persists throughout a transaction, it becomes possible to set the first the transient storage with a swap path and obtain execution within the **redeem** by specifying a fake **vault** address of the attacker.

Inside of the callback, the attacker can trigger a normal **redeem** of a smart contract, such as a queued transaction of a Gnosis Safe or from a permisionless protocol.

Because **swapForQuote** does not know that this inner **redeem** call should not use the attacker-specified swap path in transient storage, it will still be used and the attacker can take all the tokens and return nothing. The **minBuyAmount** is also always set to `0` by the vault and so there is no slippage protection there.

```
    function redeemWithRoutes(
        IAutopool vault,
        address to,
        uint256 shares,
        uint256 minAmountOut,
        ISwapRouterV2.UserSwapData[] calldata customRoutes
    ) public payable returns (uint256 amountOut) {
        ISwapRouterV2 swapRouter =
ISwapRouterV2(payable(address(systemRegistry.swapRouter())));
        swapRouter.initTransientSwap(customRoutes);

        amountOut = redeem(vault, to, shares, minAmountOut);

        //clear the routes
        swapRouter.exitTransientSwap();
        return amountOut;
    }
```

ETHERFI AND KELP DAO VAULT EXTENSIONS CAN BE DOS-ED

SEVERITY: High

REMEDIATION:

We recommend either implementing a safe mechanism to recover the reward tokens or to place the claim() function in a try catch block and remove the expected claim amount check.

STATUS: Fixed

DESCRIPTION:

The **DestinationVault** contract allows the vault manager to assign certain extensions to the vault that in their current implementation help the protocol to collect off-chain rewards from certain DAOs. Two of those are EtherFi and Kelp DAO which to collect rewards implement a merkle tree where the user by proving that that a leaf is on the merkle tree they collect the reward. To collect the extension rewards, the vault manager calls the **executeExtension()** which does a delegatecall to the extension and checks whether the balance of any tracked token has changed and if so reverts the transaction:

src\vault\DestinationVault.sol#L584-L599

```
for (uint256 i = 0; i < trackedTokensLength; ++i) {
    trackedTokensBalances[i] =
IERC20(_trackedTokens.at(i)).balanceOf(address(this));
}

// This could still set an approval that allows a later transfer out
// but that is acceptable for our use case
// slither-disable-next-line unused-return
```

```
extension.functionDelegateCall(abi.encodeCall(IDestinationVaultExtension.exe
cute, (data)));

// Verify that no tokens were pulled
for (uint256 i = 0; i < trackedTokensLength; ++i) {
    IERC20 token = IERC20(_trackedTokens.at(i));
    if (trackedTokensBalances[i] != token.balanceOf(address(this))) {
        revert ExtensionAmountMismatch();
    }
}
```

The **execute()** function that's being called as per its implementation tries to collect the rewards, swaps the reward tokens with weth and either queues the reward tokens in the **rewarder** contract or sends the rewards to the vault manager:

src\vault\extensions\base\BaseClaimingDestinationVaultExtension.sol#L70-L95

```
//
// Claim rewards
//
(uint256[] memory amountsClaimed, address[] memory tokensClaimed) =
_claim(params.claimData);

//
// Swap for reward token
//
uint256 amountReceived;
for (uint256 i = 0; i < swapParamsLength; ++i) {
    // Validations on swapData, amount returned, etc are being done in the
`BaseAsyncSwapper` level
    // solhint-disable-next-line max-line-length
    bytes memory swapData =
asyncSwapper.functionDelegateCall(abi.encodeCall(IAsyncSwapper.swap,
swapParams[i]));
    amountReceived += abi.decode(swapData, (uint256));
}
```

```
//
// Send rewards to either message sender or rewarder
//
if (params.sendToRewarder) {
    address rewarder = IDestinationVault(address(this)).rewarder();
    LibAdapter._approve(weth, rewarder, amountReceived);
    IMainRewarder(rewarder).queueNewRewards(amountReceived);
} else {
    weth.safeTransfer(msg.sender, amountReceived);
}
```

The **_claim()** function is responsible for claiming the rewards which both EtherFi and Kelp DAO extensions have nearly the same implementation with the only difference being the call arguments to the merkle tree contract. Both of the extensions receive **expectedClaimAmount** as a part of parameter which is later used to verify that the claimed amount was indeed equal to the expected claim amount:

src\vault\extensions\EtherFiClaimingDestinationVaultExtension.sol#L46-L69

```
function _claim(
    bytes memory data
) internal override returns (uint256[] memory amountsClaimed, address[]
memory tokensClaimed) {
    EtherFiClaimParams memory params = abi.decode(data,
(EtherFiClaimParams));
    uint256 expectedClaimAmount = params.expectedClaimAmount;

    Errors.verifyNotZero(expectedClaimAmount, "expectedClaimAmount");

    uint256 claimTokenBalanceBefore = claimToken.balanceOf(address(this));
    ICumulativeMerkleDrop(claimContract).claim(
        params.account, params.cumulativeAmount, params.expectedMerkleRoot,
params.merkleProof
    );
```

```
        amountsClaimed = new uint256[](1);
        tokensClaimed = new address[](1);

        amountsClaimed[0] = claimToken.balanceOf(address(this)) -
claimTokenBalanceBefore;
        tokensClaimed[0] = address(claimToken);

        // Amounts should be exact, amount we will be claiming determined before
call offchain
        if (amountsClaimed[0] != expectedClaimAmount) {
            revert InvalidAmountReceived(amountsClaimed[0],
expectedClaimAmount);
        }
    }
```

The **claim()** function can be called by anyone and the function checks that if after claiming the leaf the claimed amount hasn't changed it reverts the transaction:

https://etherscan.io/address/0x26542fbe5f320f25747e80831acdd1f27cdd0c65#code#F1#L81

```
if (preclaimed >= cumulativeAmount) revert NothingToClaim();
```

And in the case of Kelp DAO's extension it simply checks if the leaf was claimed it reverts the transaction:

https://github.com/Kelp-DAO/LRT-rsETH/blob/c2ed08e30cbba8c6d4c8e4670142fdd2009733ce/contracts/utils/MerkleDistributor/MerkleDistributor.sol#L95-L97

```
if (isClaimed(index, account)) {
    revert AlreadyClaimed();
}
```

However this creates a DoS attack vector because an attacker can front-run the **executeExtension()**, and the claim amount check in the merkle contract or the index check will revert the transaction. Even in the case that the **claim()** function is put inside of a try catch block, the transaction will still revert because of the expected claim amount check because the amount of the reward tokens would already be on the vault contract and as such the balance check will revert because the balance will stay the same.

After this two possible impacts are possible:

- If the reward token are untracked the token recovery manager can call the **recover()** function and recover the tokens
- If they are tracked the tokens can't be recovered using recover() function as it checks if the token is tracked it reverts, and the reward token can differ from the underlying token thus making it unavailable to be withdrawn using **withdrawUnderlying()**. The only way to recover those tokens would depend on the implementation of the destination vault, more specifically the implementation of the **_collectRewards()** function, which if it just sends all of the reward tokens in the vault to the vault manager or the **rewarder** it should be able to recover the tokens.

REDSTONE ORACLE AND CUSTOMSETORACLE TIME VALIDATION DIFFERENCES CAN LEAD TO ISSUES WHEN UPDATING THE PRICE

SEVERITY: Medium

REMEDIATION:

We recommend adding the same “future” checks as the Redstone oracle to avoid reverts while updating the prices.

STATUS: Fixed

DESCRIPTION:

In the **CustomRedStoneOracleAdapter.sol** when the backend calls the **updatePriceWithFeedId()** in order to update the prices in the **customOracle**, the function gets the prices of the given tokens using the Redstone oracle by calling the internal function **_securelyExtractOracleValuesAndTimestampFromTxMsg** which verifies the calldata as per the Redstone documentation. The internal function call returns two values one of them being the **timestamp** which is later verified using the **validateTimestamp()**:

src\oracles\providers\CustomRedStoneOracleAdapter.sol#L87-L90

```
(uint256[] memory values, uint256 timestamp) =
_securelyExtractOracleValuesAndTimestampFromTxMsg(feedIds);

// Call of RedstoneConsumerBase implementation of validateTimestamp
validateTimestamp(timestamp);
```

The **validateTimestamp()** function verifies that the data is not too old which is a standard check for oracles but it also verifies that the data is not from too far of a "future" which according to the code comments written by the Redstone team happens much more often than expected:

lib\redstone-evm-connector\packages\evm-connector\contracts\core\RedstoneDefaultsLib.sol#L18-L34

```
function validateTimestamp(uint256 receivedTimestampMilliseconds) internal
view {
  // Getting data timestamp from future seems quite unlikely
  // But we've already spent too much time with different cases
  // Where block.timestamp was less than dataPackage.timestamp.
  // Some blockchains may case this problem as well.
  // That's why we add MAX_BLOCK_TIMESTAMP_DELAY
  // and allow data "from future" but with a small delay
  uint256 receivedTimestampSeconds = receivedTimestampMilliseconds / 1000;

  if (block.timestamp < receivedTimestampSeconds) {
    if ((receivedTimestampSeconds - block.timestamp) >
DEFAULT_MAX_DATA_TIMESTAMP_AHEAD_SECONDS) {
      revert TimestampFromTooLongFuture(receivedTimestampSeconds,
block.timestamp);
    }
  } else if ((block.timestamp - receivedTimestampSeconds) >
DEFAULT_MAX_DATA_TIMESTAMP_DELAY_SECONDS) {
    revert TimestampIsTooOld(receivedTimestampSeconds, block.timestamp);
  }
}
```

After the Redstone library has finished validating the timestamp and the **updatePriceWithFeedId()** has finished calculating token prices the function calls the **setPrices()** function on the **customOracle** to actually update the token prices while specifying the timestamp at which the update took place which in this case is the timestamp returned by the Redstone oracle. However in the **setPrices()** the **CustomSetOracle** does additional timestamp verifications from which one checks whether the timestamp is in the future or no and if so it reverts the transaction:

src\oracles\providers\CustomSetOracle.sol#L128-130

```
if (timestamp > block.timestamp) {
    revert InvalidTimestamp(token, timestamp);
}
```

And as stated previously the Redstone oracle allows the data to be from future but because the **CustomSetOracle** reverts in the case that the data came from the future it creates a possibility that the data will not be updated because the transaction will revert. This in volatile assets can create an arbitrage opportunity or depending on how often the data comes from the “future“ it might break certain vaults due to stale prices of certain tokens that use the custom oracle.

DIRECT USAGE OF REDSTONECONSUMERBASE IN CONSUMER CONTRACT

SEVERITY: Low

PATH:

src/oracles/providers/CustomRedStoneOracleAdapter.sol#L80-L126

REMEDIATION:

See description.

STATUS: Fixed

DESCRIPTION:

The **RedstoneConsumerBase.sol** contract is intended to be a base contract for building other consumer contracts, not to be used directly in them. However, this usage is being ignored in the project's consumer contracts, which can lead to maintenance challenges and potential issues when changes are made to the underlying logic of **RedstoneConsumerBase**.

As noted in the **RedstoneConsumerBase** docs:

```
 * @dev Do not use this contract directly in consumer contracts, take a
 * look at `RedstoneConsumerNumericBase` and `RedstoneConsumerBytesBase`
instead
```

Additionally, the **_securelyExtractOracleValuesAndTimestampFromTxMsg()** function within **RedstoneConsumerBase** has a clear warning stating:

```
   * Note! You should not call this function in a consumer contract. You can
use
   * `getOracleNumericValuesFromTxMsg` or `getOracleNumericValueFromTxMsg`
instead.
   *
   * @param dataFeedIds An array of unique data feed identifiers
   * @return An array of the extracted and verified oracle values in the
same order
   * as they are requested in dataFeedIds array
   * @return dataPackagesTimestamp timestamp equal for all data packages
   */
  function _securelyExtractOracleValuesAndTimestampFromTxMsg(bytes32[]
memory dataFeedIds)
    internal
    view
    returns (uint256[] memory, uint256 dataPackagesTimestamp)
  {

  [...]
```

The **updatePriceWithFeedId()** function of the **CustomRedStoneOracleAdapter.sol** contract directly calls **_securelyExtractOracleValuesAndTimestampFromTxMsg()** for secure extraction of oracle values from the transaction calldata. This practice bypasses the recommended usage patterns intended to maintain cleaner, more maintainable code and avoid potential issues that could arise when modifications are made to **RedstoneConsumerBase**.

```
    function updatePriceWithFeedId(
        bytes32[] memory feedIds
    ) public hasRole(Roles.CUSTOM_ORACLE_EXECUTOR) {
        uint256 len = feedIds.length;
        Errors.verifyNotZero(len, "len");

        // Extract and validate the prices from the Redstone payload
@>      (uint256[] memory values, uint256 timestamp) =
_securelyExtractOracleValuesAndTimestampFromTxMsg(feedIds);
```

```
        // Call of RedstoneConsumerBase implementation of validateTimestamp
        validateTimestamp(timestamp);

        // Prepare the base tokens array
        address[] memory baseTokens = new address[](len);
        // Prepare the timestamps array and validate prices
        uint256[] memory queriedTimestamps = new uint256[](len);
        for (uint256 i = 0; i < len; ++i) {
            // Save token address from the registered mapping
            FeedId memory feedId = registeredFeedIds[feedIds[i]];
            if (feedId.tokenAddress == address(0)) {
                revert TokenNotRegistered(feedIds[i], feedId.tokenAddress);
            }
            baseTokens[i] = feedId.tokenAddress;

            // Validate the price
            Errors.verifyNotZero(values[i], "baseToken price");

            // Adjust the price from the feed decimals to 18 decimals
            uint8 feedDecimals = feedId.feedDecimals;
            if (feedDecimals < 18) {
                values[i] = values[i] * 10 ** (18 - feedDecimals);
            } else if (feedDecimals > 18) {
                values[i] = values[i] / 10 ** (feedDecimals - 18);
            }

            // Convert to ETH if the data feed price is not quoted in ETH
            if (!feedId.ethQuoted) {
                uint256 ethInUsd =
systemRegistry.rootPriceOracle().getPriceInEth(ETH_IN_USD);
                values[i] = (values[i] * 1e18) / ethInUsd;
            }

            // Set the same timestamp from the Redstone payload for all base
tokens
            queriedTimestamps[i] = timestamp / 1000; // adapted to seconds
        }
        // Set the price in the custom oracle
        customOracle.setPrices(baseTokens, values, queriedTimestamps);
    }
```

Use RedstoneConsumerNumericBase contract's getOracleNumericValuesAndTimestampFromTxMsg() function instead, as it is recommended by the Redstone Oracles team.

```
    function updatePriceWithFeedId(
        bytes32[] memory feedIds
    ) public hasRole(Roles.CUSTOM_ORACLE_EXECUTOR) {
        uint256 len = feedIds.length;
        Errors.verifyNotZero(len, "len");

        // Extract and validate the prices from the Redstone payload
--      (uint256[] memory values, uint256 timestamp) =
_securelyExtractOracleValuesAndTimestampFromTxMsg(feedIds);
++      (uint256[] memory values, uint256 timestamp) =
getOracleNumericValuesAndTimestampFromTxMsg(feedIds);

        // Call of RedstoneConsumerBase implementation of validateTimestamp
        validateTimestamp(timestamp);
[...]
```

INCENTIVE UPDATE EXTENSION EFFECTIVELY BYPASSES THE INCENTIVE CALCULATOR CHECKS

SEVERITY: Low

PATH:

src\vault\extensions\IncentiveCalculatorUpdateDestinationVaultExtension.sol#L51-L61

src\vault\DestinationVault.sol#L508-L517

REMEDIATION:

We recommend implementing the incentive calculator checks in the extension.

STATUS: Fixed

DESCRIPTION:

As explained in the issue "**TOKE4-3**" the vault allows extensions which down the line get called by the **execute()** function. In the case of **IncentiveCalculatorUpdateDestinationVaultExtension** the extension overrides the **execute()** function and allows the vault manager to overwrite the incentive calculator by manually overwriting the value stored in the slot of the **_incentiveCalculator** variable:

```
if (address(IDestinationVault(address(this)).getStats()) != oldCalc) revert
Errors.InvalidConfiguration();

address slotVal;
// slither-disable-next-line assembly
assembly {
```

```
        slotVal := sload(slot)

        if eq(slotVal, oldCalc) { sstore(slot, newCalc) }
    }

    if (address(IDestinationVault(address(this)).getStats()) != newCalc) revert
    Errors.InvalidConfiguration();
```

However the vault provides the function **setIncentiveCalculator()** which once again updates the **_incentiveCalculator** variable however there are two main differences between those two methods of updating the incentive calculator:

- The **executeExtension()** function can be called by the vault manager but the **setIncentiveCalculator()** function can be called by the member that has the **AUTO_POOL_DESTINATION_UPDATER**
- The **setIncentiveCalculator()** validates the new value of the **_incentiveCalculator** by calling the **_validateCalculator()** which each vault must implement in its own way

```
function setIncentiveCalculator(
    address incentiveCalculator_
) external hasRole(Roles.AUTO_POOL_DESTINATION_UPDATER) {
    _validateCalculator(incentiveCalculator_);

    emit IncentiveCalculatorUpdated(incentiveCalculator_);

    // slither-disable-next-line missing-zero-check
    _incentiveCalculator = incentiveCalculator_;
}
```

Those two differences create the following two issues:

- The role differences create an inconsistency in the role model of the protocol thus increasing the chances of certain members having more power than was initially delegated
- The missing validation creates a possibility that either accidentally or maliciously an incorrect calculator can be set by the vault manager which could have been stopped if the calculator was validated by the **_validateCalculator()** function

PUBLIC CONSTANTS SHOULD BE MARKED AS PRIVATE

SEVERITY: Informational

PATH:

src/stats/calculators/set/BaseSetCalculator.sol#L19

src/stats/calculators/set/StatsTransientCacheStore.sol#L19

REMEDIATION:

The mentioned variables should be marked as private instead of public.

STATUS: Fixed

DESCRIPTION:

In the following locations, there are constant variables that are declared **public**. However, setting these constants to **private** will save deployment gas. This is because the compiler won't have to create non-payable getter functions for deployment calldata, won't need to store the bytes of the value outside of where it's used, and won't add another entry to the method ID table. If necessary, the values can still be read from the verified contract source code:

```
    uint256 public constant VERSION = 2;
```

```
    uint256 public constant SET_KEYS = uint256(keccak256(bytes("SET_KEYS"))) - 1;
```

EXPLOITABLE ROUNDING ERROR IN TOKEN SWAPS

SEVERITY: Informational

PATH:

src/liquidation/BankSwapper.sol#L41-L61

REMEDIATION:

Update the buyAmountReceived calculation to always round up by using the following formula:

buyTokenAmountReceived = ((sellAmount * sellTokenPrice) + (buyTokenPrice - 1)) / buyTokenPrice;

STATUS: Fixed

DESCRIPTION:

In the **BankSwapper** contract, the **swap()** function facilitates token exchanges between the multisig **BANK** and **LiquidationRow**. Specifically, the multisig **BANK** provides a **buyToken** and receives a **sellToken**. The amount of buyToken obtained is calculated by the following formula:

```
buyTokenAmountReceived = (sellAmount * sellTokenPrice) / buyTokenPrice;
```

A rounding issue arises when (**sellTokenPrice * sellAmount**) is smaller than **buyTokenPrice**, causing **buyTokenAmountReceived** to be zero. In this scenario, the contract receives no tokens, while the **BANK** obtains the entire **sellAmount**. Consequently, it is technically possible to withdraw all contract funds through repeated use of this mechanism.

```
    function swap(
        SwapParams memory swapParams
    ) external onlyLiquidator returns (uint256 buyTokenAmountReceived) {
        IERC20 sellToken = IERC20(swapParams.sellTokenAddress);
        IERC20 buyToken = IERC20(swapParams.buyTokenAddress);
        uint256 sellAmount = swapParams.sellAmount;

        IRootPriceOracle oracle = systemRegistry.rootPriceOracle();
        uint256 sellTokenPrice =
oracle.getPriceInEth(swapParams.sellTokenAddress);
        uint256 buyTokenPrice =
oracle.getPriceInEth(swapParams.buyTokenAddress);

        // Expected buy amount from Price Oracle
        buyTokenAmountReceived = (sellAmount * sellTokenPrice) /
buyTokenPrice;

        sellToken.safeTransfer(BANK, sellAmount);
        // slither-disable-next-line arbitrary-send-erc20
        buyToken.safeTransferFrom(BANK, address(this),
buyTokenAmountReceived);

        // slither-disable-next-line reentrancy-events
        emit Swapped(address(sellToken), address(buyToken), sellAmount,
buyTokenAmountReceived, buyTokenAmountReceived);
    }
```


hexens
TOKEMAK