hexens × TOKEMAK

JULY.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 core smart contracts of Tokemak Autopilot, the second version of the Tokemak protocol. The Autopilot protocol allows liquidity providers to simply deposit their assets and let the protocol automatically optimise for the best performance and yield on those assets through highly complex strategies and models.

Our security assessment was a full review of the changes to existing smart contracts and the newly introduced smart contracts, spanning a total of 3 weeks.

During our audit, we have identified 2 medium severity vulnerabilities, various minor vulnerabilities and code optimisations.

Finally, all of our reported issues were fixed or acknowledged 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/4138bd22b4b418d929b1273887ec1e73a7a80666

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

https://github.com/Tokemak/v2-core/tree/41922dfe212637da750aa22df8ba83f089b6a04a

AUDITING DETAILS

STARTED
29.07.2024

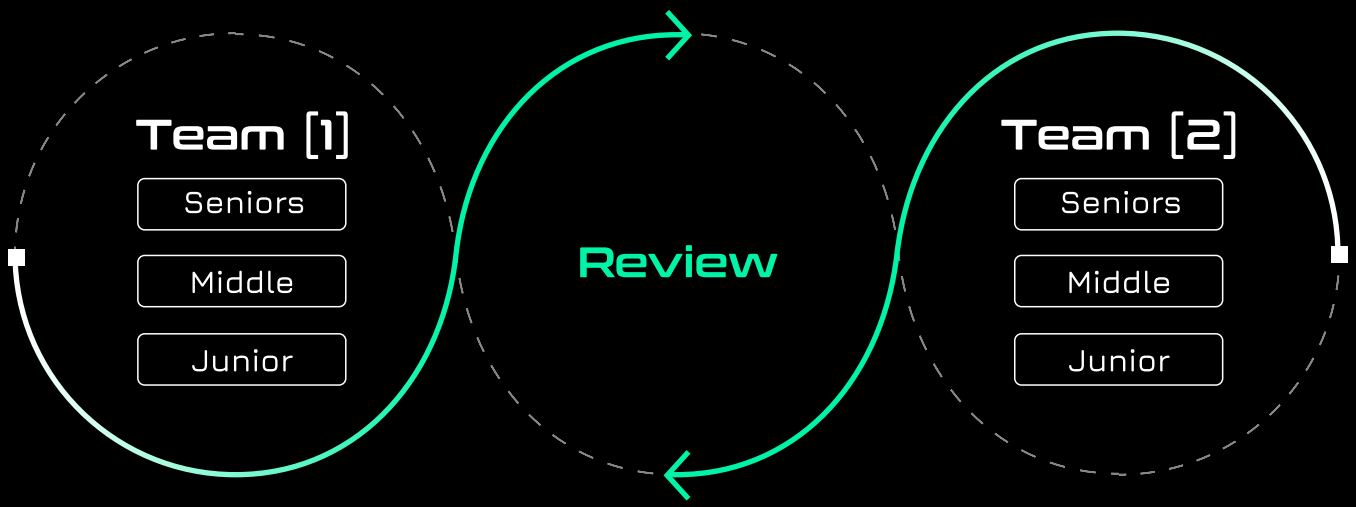
DELIVERED
20.08.2024

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.

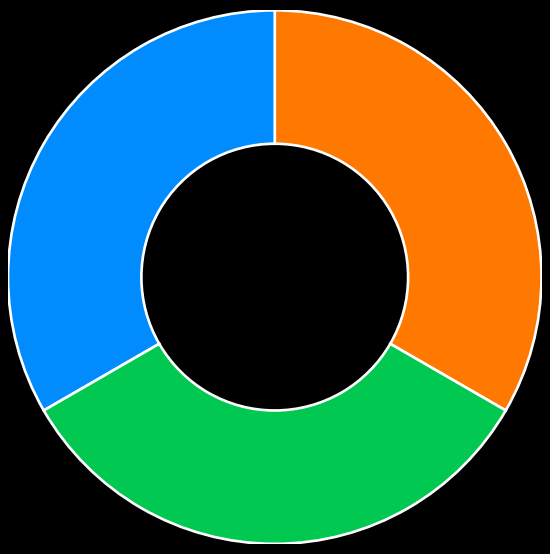
ISSUE SYMBOLIC CODES

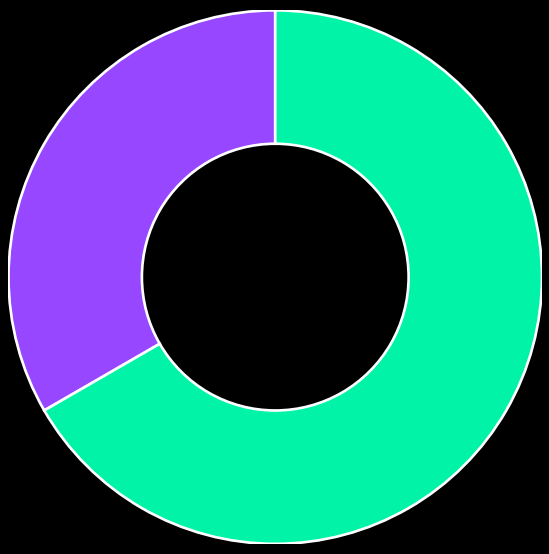
Every issue being identified and validated has its unique symbolic code assigned to the issue at the security research stage. Cause of the vulnerability reporting flow design, some of the rejected issues could be missing.

FINDINGS SUMMARY

Severity	Number of Findings
Critical	0
High	0
Medium	2
Low	2
Informational	2

Total: 6





WEAKNESSES

This section contains the list of discovered weaknesses.

TOKE2-3

BRIDGEDLSTCALCULATOR RATE CAN BE MANIPULATED THROUGH OUT-OF-ORDER MESSAGE PROCESSING

SEVERITY: Medium

PATH:

stats/calculators/bridged/BridgedLSTCalculator.sol

REMEDIATION:

When using cross-chain message we highly recommend to depend on sequence numbers instead of timestamps to preserve chronological order of messages.

STATUS: Fixed

DESCRIPTION:

Cross-chain messages are sent from EthPerTokenSender to BridgedLSTCalculator on L2 through the MessageProxy whenever there is a change in the rate of an LST on L1. This process is permissionless.

On the L2 side, the BridgedLSTCalculator only checks the timestamp of the received message to be strictly greater than the next message, while EthPerTokenSender allows for sending of multiple messages in the same block and therefore at the same timestamp.

For LST protocols that have permissionless rate updating, it could be possible to have a stale ETH share rate for some LST protocols until their next update period. For example, Stader only does update every 24 hours.

The attacker could initiate the message for the current rate, perform the rate update and then initiate another message. The new rate would be saved in EthPerTokenSender and so another call would revert (since the rate did not change) for 24 hours. However, the second message reverts on the L2 side due to the timestamp check and so the stale rate would be kept for the whole duration.

```
    function _snapshotOnMessageReceive(
        uint256 currentEthPerToken,
        uint256 snapshotTimestamp,
        uint256 newBaseApr
    ) internal {
        // Message may be retried but if the message is older than one we've
already
         // processed we don't want to accept it. The send in the same block
on the source chain
        // slither-disable-next-line timestamp
        if (lastBaseAprSnapshotTimestamp >= snapshotTimestamp) {
                        revert  OnlyNewerValue(lastBaseAprSnapshotTimestamp,
snapshotTimestamp);
        }

        [..]

    }
```

IMPROPER USE OF BLOCK.TIMESTAMP FOR DEADLINE

SEVERITY: Medium

PATH:

src/destinations/adapters/AerodromeAdapter.sol::_runWithdrawal()#L83-L106

REMEDIATION:

The deadline parameter should be implemented so that users can specify their desired deadline.

STATUS: Fixed

DESCRIPTION:

In the **AerodromeAdapter.sol** contract, the **IRouter.removeLiquidity()** function sets the **deadline** parameter for liquidity removal to **block.timestamp**. This can lead to issues because miners have some control over **block.timestamp**, which may result in time-based manipulation. Furthermore, users are not given the option to specify their own deadline, which is a key feature for ensuring transaction validity within a certain time window:

```
    function _runWithdrawal(AerodromeRemoveLiquidityParams memory _params)
        internal
        returns (uint256[] memory actualAmounts, uint256 lpBurnAmount)
    {
        uint256 lpTokensBefore =
IERC20(_params.pool).balanceOf(address(this));
        (uint256 amountA, uint256 amountB) =
IRouter(_params.router).removeLiquidity(
            _params.tokens[0],
            _params.tokens[1],
            _params.stable,
            _params.maxLpBurnAmount,
            _params.amounts[0],
            _params.amounts[1],
            address(this),
>>          block.timestamp
        );

        uint256 lpTokensAfter =
IERC20(_params.pool).balanceOf(address(this));

        lpBurnAmount = lpTokensBefore - lpTokensAfter;

        actualAmounts = new uint256[](2);
        actualAmounts[0] = amountA;
        actualAmounts[1] = amountB;
    }
```

In contrast, the **MaverickAdapter.sol** contract handles this situation correctly by allowing the user to pass a **deadline** as a parameter in the **MaverickWithdrawalExtraParams** struct. This provides users with greater flexibility and protection against potential vulnerabilities:

```
    function _runWithdrawal(
        IRouter router,
        uint256[] calldata amounts,
        MaverickWithdrawalExtraParams memory maverickExtraParams
        ) private returns (uint256 tokenAAmount, uint256 tokenBAmount,
IPool.BinDelta[] memory binDeltas) {
        (tokenAAmount, tokenBAmount, binDeltas) = router.removeLiquidity(
            IPool(maverickExtraParams.poolAddress),
            address(this),
            maverickExtraParams.tokenId,
            maverickExtraParams.maverickParams,
            amounts[0],
            amounts[1],
            maverickExtraParams.deadline
        );
    }
```

REDUNDANT INITIALIZATION OF ZEROADDRESSREACHED IN _VALIDATEHOOKS() FUNCTION

SEVERITY: Low

PATH:

src/strategy/AutopoolETHStrategyConfig.sol#L235

REMEDIATION:

There is no need to initialize zeroAddressReached with the value false

- bool zeroAddressReached = false;
+ bool zeroAddressReached;

STATUS: Fixed

DESCRIPTION:

In the function **AutoPoolETHStrategyConfig::_validateHooks()**, the variable **zeroAddressReached** is initialized with the value **false**. However, the default value of a new boolean variable is always **false**, making this initialization redundant.

```
    bool zeroAddressReached = false;
```

TOKE2-6

LACK OF DYNAMIC PRICE STALE CHECK ADJUSTMENT MECHANISM

SEVERITY: Low

PATH:

src/stats/calculators/AerodromeStakingIncentiveCalculator.sol

REMEDIATION:

Consider implementing a setter function for the PRICE_STALE_CHECK parameter. This function should only be callable by a trusted entity (such as the owner or a governance contract) to prevent abuse.

STATUS: Acknowledged

DESCRIPTION:

The **PRICE_STALE_CHECK** parameter in the **AerodromeStakingIncentiveCalculator.sol** contract is currently hardcoded to a fixed value of **12 hours**. This design choice may hinder the protocol's ability to react to volatile market conditions effectively.

```
    /// @dev Duration after which a price/data becomes stale.
    uint40 public constant PRICE_STALE_CHECK = 12 hours;
```

As market dynamics change, especially in volatile DeFi ecosystems, the rigidity of this fixed parameter may lead to inaccurate incentive pricing, liquidity provider token prices, and staking APR calculations. Consequently, outdated price data could be used during key contract operations, potentially leading to suboptimal reward distributions or incorrect incentive calculations.

```
    function _getIncentivePrice(address _token) internal view returns
(uint256) {
        IIncentivesPricingStats pricingStats =
systemRegistry.incentivePricing();
        (uint256 fastPrice, uint256 slowPrice) =
pricingStats.getPrice(_token, PRICE_STALE_CHECK);
        return Math.min(fastPrice, slowPrice);
    }
```

INCONSISTENT SCRIPT NAMING CONVENTIONS

SEVERITY: Informational

REMEDIATION:

All script files should follow the .s.sol naming convention to align with Foundry's standards and ensure clarity between contracts, libraries, and scripts.

STATUS: Acknowledged

DESCRIPTION:

In the project, there are several scripts incorrectly named with the **.sol** extension instead of the expected **.s.sol** suffix. This inconsistency can cause confusion and prevent Forge from recognizing these files as scripts, potentially leading to build or execution issues.

```
script/base/02_DestinationTemplatesSetup.sol
script/base/06_StatBridging.sol
script/base/07_StatBridgingSet2.sol
script/mainnet/02_DestinationTemplatesSetup.sol
script/mainnet/10_Maverick.sol
script/mainnet/11_StatBridging.sol
script/mainnet/12_StatBridgingBaseReceipt.sol
script/mainnet/13_StatBridgingBaseSet2.sol
script/mainnet/14_ChainlinkPerTokenSender.sol
script/sepolia/04_Router.sol
script/utils/Constants.sol
```

OPTIMIZATION OF ZERO TVL CHECK IN CALCULATEETHPERTOKEN FUNCTION

SEVERITY: Informational

PATH:

src/stats/calculators/EzethLRTCalculator.sol::calculateEthPerToken()#L53-L69

REMEDIATION:

See description.

STATUS: Fixed

DESCRIPTION:

The current implementation of the **calculateEthPerToken()** function includes a conditional check for a **totalTVL** value of zero, which returns 0 when detected:

```
        if (totalTVL == 0) {
            return 0;
        }
```

However, the likelihood of **totalTVL** being zero is extremely low due to the constant accumulation of staked assets in the protocol. If **totalTVL** were zero, the existing calculation logic would naturally return 0 in the final division step, making this explicit check redundant.

```
function calculateEthPerToken() public view override returns (uint256) {
    // Get the total TVL priced in ETH from restakeManager
    // slither-disable-next-line unused-return
    (,, uint256 totalTVL) = renzoRestakeManger.calculateTVLs();

    uint256 totalSupply = IERC20(lstTokenAddress).totalSupply();

    if (totalSupply == 0) {
        return 1e18;
    }

    if (totalTVL == 0) {
        return 0;
    }

    return (10 ** 18 * totalTVL) / totalSupply;
}
```

Remove the redundant check:

```
    function calculateEthPerToken() public view override returns (uint256) {
        // Get the total TVL priced in ETH from restakeManager
        // slither-disable-next-line unused-return
        (,, uint256 totalTVL) = renzoRestakeManger.calculateTVLs();

        uint256 totalSupply = IERC20(lstTokenAddress).totalSupply();

        if (totalSupply == 0) {
            return 1e18;
        }

--      if (totalTVL == 0) {
--          return 0;
--      }

        return (10 ** 18 * totalTVL) / totalSupply;
    }
```


hexens
TOKEMAK