hexens × Valantis

SEPT.24

SECURITY REVIEW REPORT FOR VALANTIS

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.

SCOPE

The analyzed resources are located on:

https://github.com/ValantisLabs/Validly/tree/a767baff3fbdf114f74798926fd838e744bb5c01

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

https://github.com/ValantisLabs/Validly/commit/858e116264511833669a053e0af32b25ef2dc7e6

AUDITING DETAILS

STARTED
09.09.2024

DELIVERED
18.09.2024

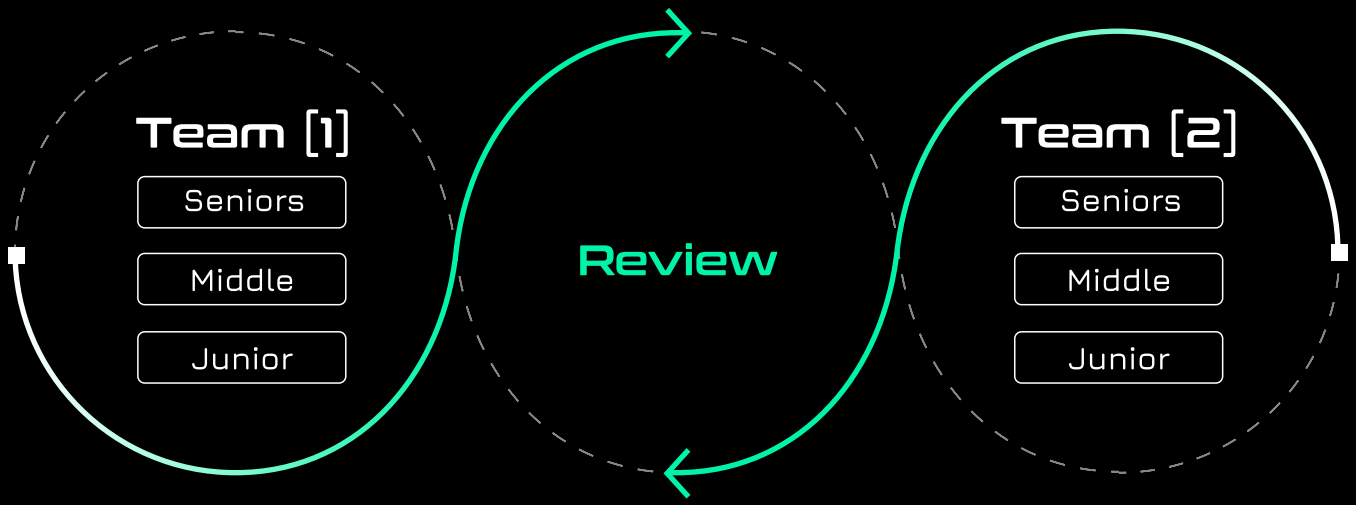
Review Led by

TRUNG DINH

Senior Security Researcher | 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.

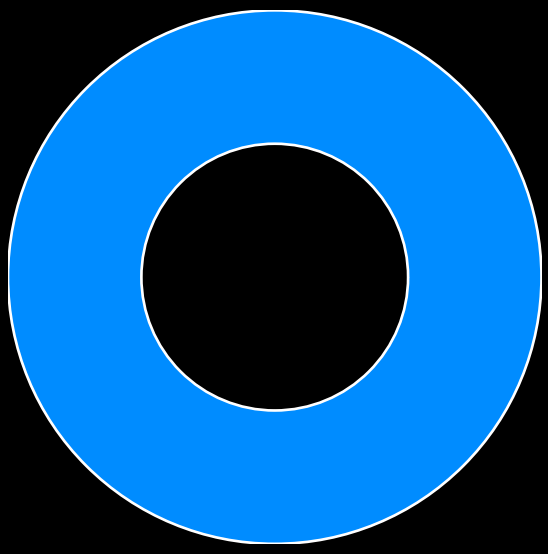
ISSUE SYMBOLIC CODES

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.

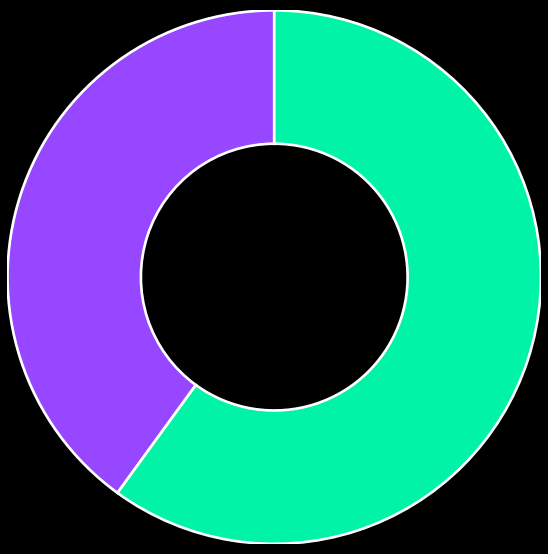
FINDINGS SUMMARY

Severity	Number of Findings
Critical	0
High	0
Medium	0
Low	0
Informational	5

Total: 5



- Informational



- Fixed
- Acknowledged

WEAKNESSES

This section contains the list of discovered weaknesses.

VLTS2-1

UNUSED ERROR

SEVERITY: Informational

PATH:

Validly.sol#L29

REMEDIATION:

If the error is redundant and there is no missing logic where it can be used, it should be removed.

STATUS: Fixed

DESCRIPTION:

In the **Validly.sol** contract the error **Validly__priceOutOfRange** on line 29, is declared but never used.

```
error Validly__priceOutOfRange();
```

CONSTANT VARIABLES SHOULD BE MARKED AS PRIVATE

SEVERITY: Informational

PATH:

src/Validly.sol#L51-L80

src/ValidlyFactory.sol#L34-L41

REMEDIATION:

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

STATUS: Acknowledged

DESCRIPTION:

In mentioned contracts, there are **constant** and **immutable** variables that are declared **public**. However, setting **constants** and **immutables** 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 MINIMUM_LIQUIDITY = 1000;

    ISovereignPool public immutable pool;

    bool public immutable isStable;

    uint256 public immutable decimals0;

    uint256 public immutable decimals1;
```

```
IProtocolFactory public immutable protocolFactory;

uint256 public immutable feeBips;
```

THE VALIDLYFACTORY CONTRACT SHOULD INCLUDE MORE FUNCTIONS TO INTERACT WITH THE SOVEREIGN POOL

SEVERITY: Informational

PATH:

src/ValidlyFactory.sol

REMEDIATION:

Consider adding more functions for the ValidlyFactory contract to interact with the SovereignPool contract.

STATUS: Acknowledged

DESCRIPTION:

The **ValidlyFactory.createPair()** function deploys a new instance of the Sovereign Pool, passing **address(this)** (the ValidlyFactory itself) to the constructor as the **poolManager**. This gives the ValidlyFactory control over pool manager-restricted functions, such as:

- **setPoolManager()**
- **setPoolManagerFeeBips()**
- **setSovereignOracle()**
- **setSwapFeeModule()**
- **claimPoolManagerFees()**

However, the **ValidlyFactory** contract currently only implements **claimFees()** and **setPoolManagerFeeBips()** to interact with the pool's **claimPoolManagerFees()** and **setPoolManagerFeeBips()**. It lacks functions to interact with the other critical pool configuration functions. This omission may prevent the pool from updating settings, such as changing the oracle, when needed.

```
    SovereignPoolConstructorArgs memory args = SovereignPoolConstructorArgs(
        _token0,
        _token1,
        address(protocolFactory),
        address(this),
        address(0),
        address(0),
        false,
        false,
        0,
        0,
        feeBips
    );
```

VLTS2-4

REDUNDANT ADDRESS(0) CHECK IN VALIDLY.DEPOSIT() FUNCTION

SEVERITY: Informational

PATH:

src/Validly.sol#L149-L151

REMEDIATION:

See description.

STATUS: Fixed

DESCRIPTION:

In the **Validly.deposit()** function, there is a check to revert the transaction if the **_recipient** address is **address(0)**. However, this check is unnecessary, as the internal **_mint()** function in the ERC20 contract already performs the same check, ensuring that the recipient address is not **address(0)**.

```
abstract contract ERC20 is Context, IERC20, IERC20Metadata, IERC20Errors {
    ...
    function _mint(address account, uint256 value) internal {
        if (account == address(0)) {
            revert ERC20InvalidReceiver(address(0));
        }
        _update(address(0), account, value);
    }
    ...
}
```

```
if (_recipient == address(0)) {
    revert Validly__deposit_invalidRecipient();
}
```

Consider removing the address(0) check for the _recipient address.

```
function deposit(
    uint256 _amount0,
    uint256 _amount1,
    uint256 _minShares,
    uint256 _deadline,
    address _recipient,
    bytes calldata _verificationContext
)
    external
    override
    ensureDeadline(_deadline)
    nonReentrant
    returns (uint256 shares, uint256 amount0, uint256 amount1)
{
-    if (_recipient == address(0)) {
-        revert Validly__deposit_invalidRecipient();
-    }
```

VLTS2-5

THE COMMENT IN THE VALIDLY._GET_Y_STABLEINVARIANT() FUNCTION IS INCORRECT

SEVERITY: Informational

PATH:

src/Validly.sol#L397-L398

REMEDIATION:

Consider fixing the comment to:

- Did not converge in 256 fixed point iterations
+ Did not converge in 255 fixed point iterations

STATUS: Fixed

DESCRIPTION:

The function **Validly._get_y_stableInvariant** loops from **0** to **254**, resulting in a total of 255 iterations. However, the comment states, "Did not converge in 256 fixed point iterations," which does not match the actual 255 iterations.

```
function _get_y_stableInvariant(uint256 x0, uint256 invariant, uint256 y)
private pure returns (uint256) {
    for (uint256 i = 0; i < 255; i++) {
        uint256 y_prev = y;
        uint256 k = _f(x0, y);
        if (k < invariant) {
            uint256 dy = ((invariant - k) * 1e18) / _d(x0, y);
            y = y + dy;
        } else {
            uint256 dy = ((k - invariant) * 1e18) / _d(x0, y);
            y = y - dy;
        }
        if (y > y_prev) {
            if (y - y_prev <= 1) {
                return y;
            }
        } else {
            if (y_prev - y <= 1) {
                return y;
            }
        }
    }
    // Did not converge in 256 fixed point iterations,
    // revert for safety
    revert Validly___get_y_notConverged();
}
```


hexens
×
Valantis