hexens × Swell

MAR.24

SECURITY REVIEW REPORT FOR SWELL

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.

CONTENTS

EXECUTIVE SUMMARY

OVERVIEW

This audit covered Swell Network's CumulativeMerkleDrop contract, which handles the initial distribution of the SWELL token using Merkle proofs, as well as the Staking contract where users can simply stake their SWELL tokens.

Our security assessment was a full review of the two smart contracts, spanning a total of 3 days.

During our audit we have 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/SwellNetwork/swell-contracts/tree/2559c9e4f6140f6080bc0e3a2dc94410c82ac907

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

https://github.com/SwellNetwork/swell-contracts/tree/b13923d8c4a979bcfa5ceec9b35f65137e2e7634

AUDITING DETAILS

STARTED
10.03.2024

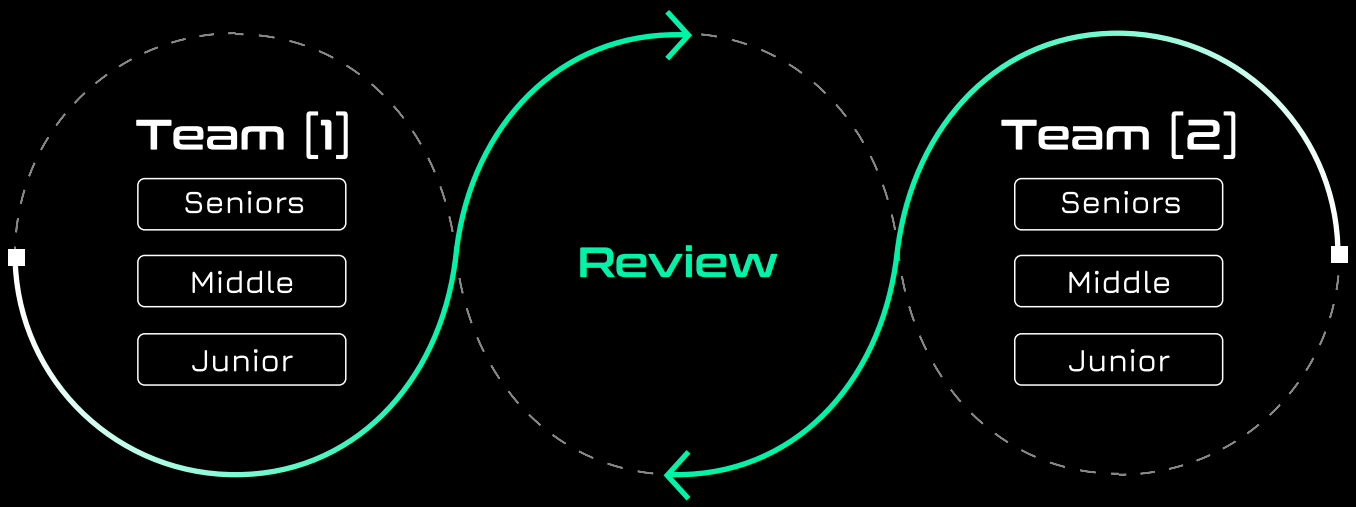
DELIVERED
12.03.2024

Review Led by

KASPER ZWIJSEN
Head of Smart Contract 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.

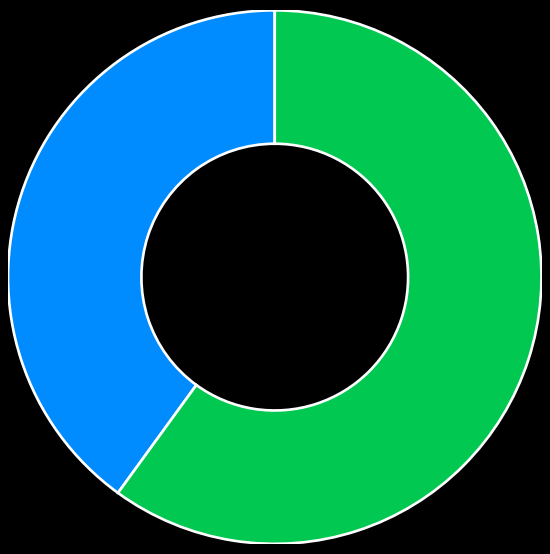
ISSUE SYMBOLIC CODES

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.

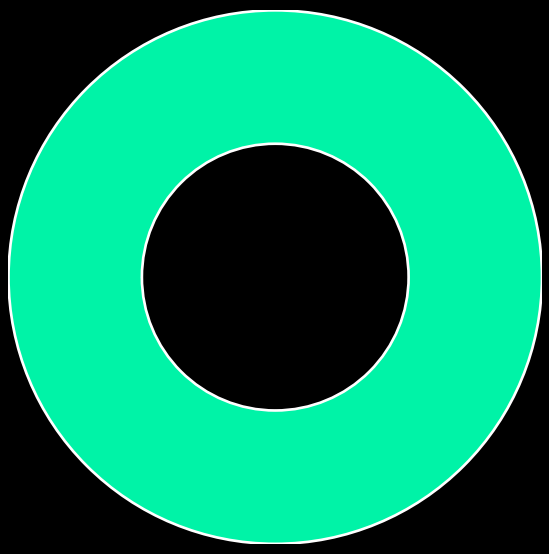
FINDINGS SUMMARY

Severity	Number of Findings
Critical	0
High	0
Medium	0
Low	3
Informational	2

Total: 5



- Low
- Informational



- Fixed

WEAKNESSES

This section contains the list of discovered weaknesses.

SWSMC-1

CHANGING THE COOLDOWN TIME MAY RESULT IN FUNDS BEING UNEXPECTEDLY LOCKED FOR A LONGER TIME

SEVERITY: Low

PATH:

Staking.sol:unlock()

REMEDIATION:

See description.

STATUS: Fixed

DESCRIPTION:

There exists an edge case for the unlocking of funds, where an unlock request funds might persist for a longer time (based on the old **cooldownTime**) in the scenario described below, as the cycle within the **unlock()** function terminates when **requests[i].unlockTime > block.timestamp**.

1. Let's assume that **cooldownTime** is 7 days.
2. **user** locks 1000 tokens.

3. **used** calls **requestUnlock(300)** to request 300 token unlock, **so unlockRequests[msg.sender].unlockTime** is **block.timestamp + cooldownTime** (current timestamp plus 7 days).
4. **owner** changes **cooldownTime** to 1 day.
5. **user** calls **requestUnlock(700)** to request 700 token unlock, so **unlockRequests[msg.sender].unlockTime** is **block.timestamp + cooldownTime** (current timestamp plus 1 day).
6. After passing 1 day user calls **unlock()** function to unlock his 700 tokens.
7. Because the first request with index 0's **unlockTime** is later than current timestamp (**requests[i].unlockTime > block.timestamp**) the cycle inside **unlock** function will break, and the **user** should wait at least 7 days to unlock his tokens, even if the **unlockTime** of the second request has already expired.

```
function lock(address _account, uint256 _amount) external {
    if (_account == address(0)) revert ADDRESS_NULL();
    if (_amount == 0) revert AMOUNT_NULL();

    ERC20(TOKEN).transferFrom(msg.sender, address(this), _amount);

    balanceOf[_account] += _amount;
    totalSupply += _amount;

    emit Lock(_account, _amount);
}

/// @inheritdoc IStaking
function requestUnlock(uint256 _amount) external {
    if (_amount == 0) revert AMOUNT_NULL();
    if (balanceOf[msg.sender] < _amount) revert INSUFFICIENT_BALANCE();
    if (unlockRequests[msg.sender].length >= MAX_REQUESTS) revert
MAX_REQUESTS_REACHED();

    balanceOf[msg.sender] -= _amount;
    totalSupply -= _amount;

    unlockRequests[msg.sender].push(Request(_amount, block.timestamp +
cooldownTime));

    emit RequestUnlock(msg.sender, _amount, block.timestamp + cooldownTime);
}

/// @inheritdoc IStaking
function unlock() external {
    Request[] storage requests = unlockRequests[msg.sender];

    uint256 total;
    uint256 len = requests.length;
    for (uint256 i; i < len; ++i) {
        if (requests[i].unlockTime == 0) continue;
        if (requests[i].unlockTime <= block.timestamp) {
            total += requests[i].amount;
            delete requests[i];
        } else {
            break;
```

```
        }
    }

    if (total > 0) ERC20(TOKEN).transfer(msg.sender, total);

    emit Unlock(msg.sender, total);
}
```

We would recommend allowing a user not to break the loop inside the **unlock()** function, based on a parameter, so breaking early is still possible as gas optimisation:

```
function unlock(bool stopEarly) external {
    Request[] storage requests = unlockRequests[msg.sender];

    uint256 total;
    uint256 len = requests.length;
    for (uint256 i; i < len; ++i) {
        if (requests[i].unlockTime == 0) continue;
        if (requests[i].unlockTime <= block.timestamp) {
            total += requests[i].amount;
            delete requests[i];
        } else if (stopEarly) {
            break;
        }
    }

    if (total > 0) ERC20(TOKEN).transfer(msg.sender, total);

    emit Unlock(msg.sender, total);
}
```

GAS INEFFICIENT WAY OF HANDLING USER REQUESTS

SEVERITY: Low

PATH:

Staking.sol

REMEDIATION:

We would recommend to implement a more gas efficient way of unlocking and removing a user's unlock requests.

For example by using a mapping instead of array and a linearly incremental pointer to the first valid unlock request in that mapping, as well as the current length.

requestUnlock would increment the length (totalUnlockRequests[msg.sender]) and store the unlock request in the mapping at that length.

unlock can loop from the pointer to the length (and possibly break earlier), while still deleting the mapping entries (for refund) and at the end simply increasing the pointer with the amount of unlocked requests. The next call would not have to loop over unlocked requests.

cleanEmptyRequests can be removed.

STATUS: Fixed

DESCRIPTION:

A user has to create an unlock request using **requestUnlock**, which will push a new unlock request to the array, keeping the order of timestamps.

The order of timestamps allows for a more gas efficient unlocking in **unlock**, however this function only deletes the data in the array entry upon unlock. Subsequent unlocks will still loop through these empty elements, which

costs an extra **SLOAD** for each entry each time.

To prevent this, a user is supposed to call **cleanEmptyRequests**, which shifts the pending requests to the front and pops the empty requests. This can still cost a significant amount of gas, depending on the amount of empty requests, where fewer will have a higher gas cost.

So in both cases, the user will be paying more gas due to the gas inefficient way handling these requests.

```
function requestUnlock(uint256 _amount) external {
    if (_amount == 0) revert AMOUNT_NULL();
    if (balanceOf[msg.sender] < _amount) revert INSUFFICIENT_BALANCE();
    if (unlockRequests[msg.sender].length >= MAX_REQUESTS) revert
MAX_REQUESTS_REACHED();

    balanceOf[msg.sender] -= _amount;
    totalSupply -= _amount;

    unlockRequests[msg.sender].push(Request(_amount, block.timestamp +
cooldownTime));

    emit RequestUnlock(msg.sender, _amount, block.timestamp + cooldownTime);
}

/// @inheritdoc IStaking
function unlock() external {
    Request[] storage requests = unlockRequests[msg.sender];

    uint256 total;
    uint256 len = requests.length;
    for (uint256 i; i < len; ++i) {
        if (requests[i].unlockTime == 0) continue;
        if (requests[i].unlockTime <= block.timestamp) {
            total += requests[i].amount;
            delete requests[i];
        } else {
            break;
        }
    }

    if (total > 0) ERC20(TOKEN).transfer(msg.sender, total);

    emit Unlock(msg.sender, total);
}

/// @inheritdoc IStaking
function cleanEmptyRequest() external {
    Request[] storage requests = unlockRequests[msg.sender];
    Request[] memory refRequests = requests;
```

```
    uint256 emptySlots;
    uint256 len = refRequests.length;
    // Count the number of empty slots at the beginning of the array
    for (uint256 i; i < len; ++i) {
        if (refRequests[i].unlockTime == 0) ++emptySlots;
        else break;
    }

    // Shift the elements to the left to override the empty slots
    for (uint256 i; i < len - emptySlots; ++i) {
        requests[i] = refRequests[i + emptySlots];
    }

    // Pop as much as empty slots
    for (uint256 i; i < emptySlots; ++i) {
        requests.pop();
    }
}
```

SWSMC-6

LACK OF UPPER LIMIT CHECK FOR LOCK DURATION ALLOWS FREEZING OF ASSETS

SEVERITY: Low

REMEDIATION:

Sensible upper limits should be checked before setting the value.

STATUS: Fixed

DESCRIPTION:

In the **Staking** contract, **cooldownTime** represents the duration one must wait after requesting an unlock. While this variable is modifiable, the corresponding setter function lacks an upper-bound check for the values it accepts. Consequently, **cooldownTime** can be to any arbitrary **uint256** value, thereby preventing the unlocking of user assets.

```
function setCooldownDuration(uint256 _duration) external onlyOwner {
        emit CooldownUpdated(cooldownTime, _duration);

        cooldownTime = _duration;
    }
```

SWSMC-3

GAS IMPROVEMENTS FOR ARITHMETIC OPERATIONS

SEVERITY: Informational

PATH:

Staking.sol, CumulativeMerkleDrop.sol

REMEDIATION:

Relocate all arithmetic operations inside an unchecked block.

STATUS: Fixed

DESCRIPTION:

Since there's no possibility of overflow/underflow within the contracts, it's advisable to relocate all arithmetic operations inside an unchecked block.

Steking.sol

```
/// @inheritdoc IStaking
function lock(address _account, uint256 _amount) external {
    if (_account == address(0)) revert ADDRESS_NULL();
    if (_amount == 0) revert AMOUNT_NULL();

    ERC20(TOKEN).transferFrom(msg.sender, address(this), _amount);

    balanceOf[_account] += _amount;
    totalSupply += _amount;

    emit Lock(_account, _amount);
}
```

```
/// @inheritdoc IStaking
function requestUnlock(uint256 _amount) external {
    if (_amount == 0) revert AMOUNT_NULL();
    if (balanceOf[msg.sender] < _amount) revert INSUFFICIENT_BALANCE();
    if (unlockRequests[msg.sender].length >= MAX_REQUESTS) revert
MAX_REQUESTS_REACHED();

    balanceOf[msg.sender] -= _amount;
    totalSupply -= _amount;

    unlockRequests[msg.sender].push(Request(_amount, block.timestamp +
cooldownTime));

    emit RequestUnlock(msg.sender, _amount, block.timestamp + cooldownTime);
}

/// @inheritdoc IStaking
function unlock() external {
    Request[] storage requests = unlockRequests[msg.sender];

    uint256 total;
    uint256 len = requests.length;
    for (uint256 i; i < len; ++i) {
        if (requests[i].unlockTime == 0) continue;
        if (requests[i].unlockTime <= block.timestamp) {
            total += requests[i].amount;
            delete requests[i];
        } else {
            break;
        }
    }

    if (total > 0) ERC20(TOKEN).transfer(msg.sender, total);

    emit Unlock(msg.sender, total);
}
```

```
/// @inheritdoc IStaking
function cleanEmptyRequest() external {
    Request[] storage requests = unlockRequests[msg.sender];
    Request[] memory refRequests = requests;

    uint256 emptySlots;
    uint256 len = refRequests.length;
    // Count the number of empty slots at the beginning of the array
    for (uint256 i; i < len; ++i) {
        if (refRequests[i].unlockTime == 0) ++emptySlots;
        else break;
    }

    // Shift the elements to the left to override the empty slots
    for (uint256 i; i < len - emptySlots; ++i) {
        requests[i] = refRequests[i + emptySlots];
    }

    // Pop as much as empty slots
    for (uint256 i; i < emptySlots; ++i) {
        requests.pop();
    }
}
```

CumulativeMerkleDrop.sol

```
/// @inheritdoc ICumulativeMerkleDrop
function claimAndLock(uint256 cumulativeAmount, uint256 amountToLock,
uint256 index, bytes32[] calldata merkleProof)
    external
    onlyClaimOpen
{
    // Verify the merkle proof
    if (!verifyProof(merkleProof, index, cumulativeAmount, msg.sender))
revert INVALID_PROOF();

    // Mark it claimed
    uint256 preclaimed = cumulativeClaimed[msg.sender];
    if (preclaimed >= cumulativeAmount) revert NOTHING_TO_CLAIM();
    cumulativeClaimed[msg.sender] = cumulativeAmount;

    // Send the token
    uint256 amount = cumulativeAmount - preclaimed;
    if (amountToLock > amount) revert INSUFFICIENT_AMOUNT();
    if (amountToLock > 0) _lock(amountToLock);
    if (amount != amountToLock) ERC20(token).transfer(msg.sender, amount -
amountToLock);

    emit Claimed(msg.sender, amount);
}
```

SWSMC-5

CUSTOM ERROR

SEVERITY: Informational

REMEDIATION:

We would recommend replacing those checks with a custom error in favour of saving deployment gas and byte code size.

STATUS: Fixed

DESCRIPTION:

In the **CumulativeMerkleDrop** contract, validation checks are executed through custom errors, with the require statement employed exclusively at a single location.

```
function setClaimStatus(uint8 status) external onlyOwner {
        require(status == 1 || status == 2, "Invalid status");
        claimIsOpen = status;
    }
```


hexens
×
Swell