hexens × Swell

APR.24

SECURITY REVIEW REPORT FOR SWELL

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 the L2 Pre-Deposit contracts for Swell. The contracts included a simple staking contract for WETH, wstETH and weETH, as well as a zap contract that interacts with the staking contract for the user by wrapping and staking ETH, stETH and eETH.

Our security assessment was a full review of the 2 new smart contracts, spanning a total of 2 days.

During our audit, we did not identify any high severity vulnerabilities. We did identify various minor issues 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/pull/9

Branch: feat/staking

Commit hash: 5b421683380c3aa4077454ce6177f3d3ac24cb9b

SimpleStakingERC20.sol and Zap.sol.

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

b30a92b068d5875e86ce5f0f34248b1506e1c343

AUDITING DETAILS

STARTED
02.04.2024

DELIVERED
03.04.2024

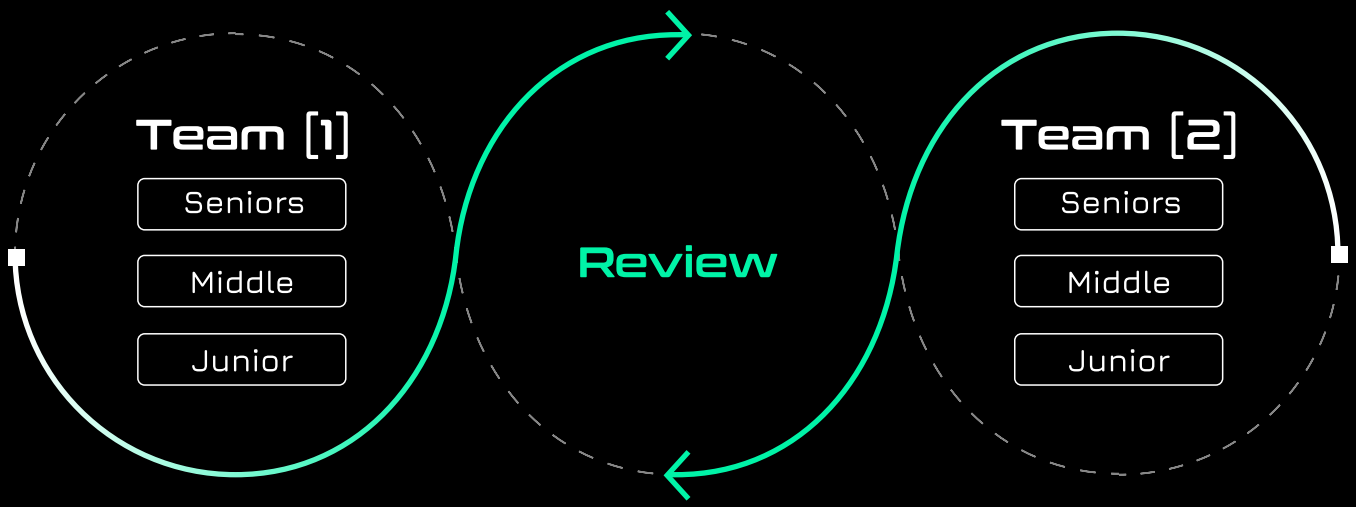
Review Led by

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.

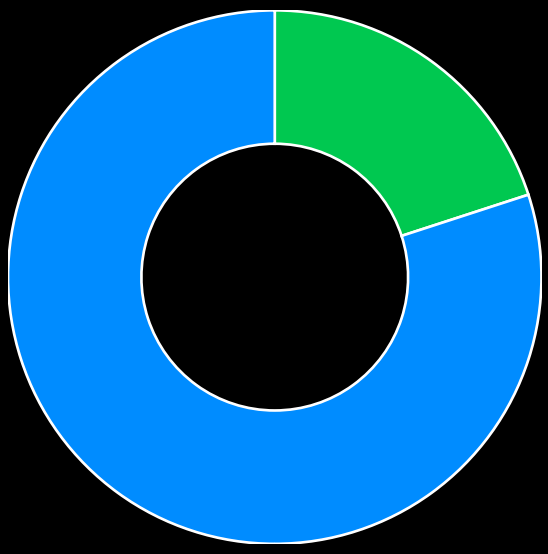
ISSUE SYMBOLIC CODES

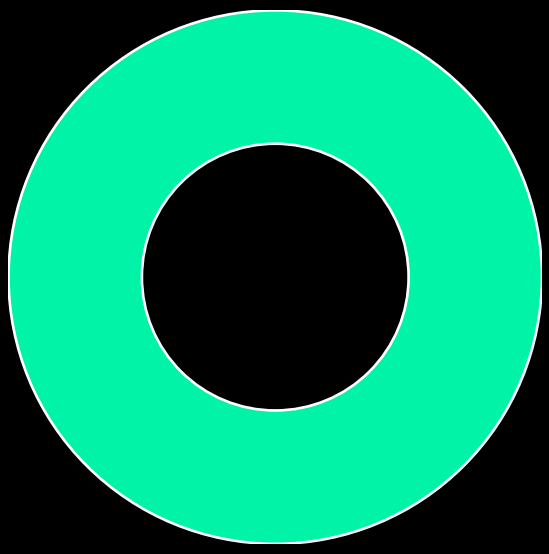
Every issue being identified and validated has its unique symbolic code assigned to the issue at the security research stage. Cause of the vulnerability reporting flow design, some of the rejected issues could be missing.

FINDINGS SUMMARY

Severity	Number of Findings
Critical	0
High	0
Medium	0
Low	1
Informational	4

Total: 5





WEAKNESSES

This section contains the list of discovered weaknesses.

SWLLL2-4

ZAP BALANCE CHECK OPTIMISATION

SEVERITY: Low

PATH:

src/Zap.sol:stETHZapIn, eETHZapIn (L42-50, L52-60)

REMEDIATION:

See description.

STATUS: Fixed

DESCRIPTION:

In the Zap contract, there are functions to take stETH and eETH. It will transfer the amounts from the user and wrap it in the wrapped variant before calling the deposit function of the stake contract.

However, the Zap contract is not supposed to hold funds and so a double **balanceOf** to handle deflationary tokens (or actually it is the share rate rounding of stETH that causes 1-2 wei to go missing) is not necessary.

Instead, the first **balanceOf** should be removed and after the transfer, the function should simply stake the entire balance of the contract. This saves a **STATICCALL** on every Zap deposit call.

```
function stETHZapIn(uint256 _amount) external {
    // Transfer stETH from msg.sender to this contract, sometimes 1 or 2 wei can
be missing.
    uint256 bal = stETH.balanceOf(address(this));
    stETH.transferFrom(msg.sender, address(this), _amount);
    bal = stETH.balanceOf(address(this)) - bal;

    // Deposit wstETH to staking contract
    stakingContract.deposit(ERC20(address(wstETH)), wstETH.wrap(bal),
msg.sender);
}

function eETHZapIn(uint256 _amount) external {
    // Transfer eETH from msg.sender to this contract, sometimes 1 or 2 wei can
be missing.
    uint256 bal = eETH.balanceOf(address(this));
    eETH.transferFrom(msg.sender, address(this), _amount);
    bal = eETH.balanceOf(address(this)) - bal;

    // Deposit eETH to staking contract
    stakingContract.deposit(ERC20(address(weETH)), weETH.wrap(bal), msg.sender);
}
```

For example:

```
function stETHZapIn(uint256 _amount) external {
    // Transfer stETH from msg.sender to this contract, sometimes 1 or 2 wei can
be missing.
    stETH.transferFrom(msg.sender, address(this), _amount);
    uint256 bal = stETH.balanceOf(address(this));

    // Deposit wstETH to staking contract
    stakingContract.deposit(ERC20(address(wstETH)), wstETH.wrap(bal),
msg.sender);
}

function eETHZapIn(uint256 _amount) external {
    // Transfer eETH from msg.sender to this contract, sometimes 1 or 2 wei can
be missing.
    eETH.transferFrom(msg.sender, address(this), _amount);
    uint256 bal = eETH.balanceOf(address(this));

    // Deposit eETH to staking contract
    stakingContract.deposit(ERC20(address(weETH)), weETH.wrap(bal), msg.sender);
}
```

SWLLL2-1

IMPLEMENTATION CONTRACT IMPORTS

SEVERITY: Informational

PATH:

src/Zap.sol:L4-5

src/SimpleStakingERC20.sol:L5

REMEDIATION:

Replace ERC20 and WETH with IERC20 and IWETH for the imports and uses in the code.

STATUS: Fixed

DESCRIPTION:

In the Zap and SimpleStakingERC20 contracts, the implementation versions of ERC20 and WETH are imported but the contracts themselves do not create any new tokens. They only use external functions of these contracts and so it would be best practise style wise that the interfaces IERC20 and IWETH should be imported and used instead.

```
import {WETH} from "solmate/tokens/WETH.sol";
import {ERC20} from "solmate/tokens/ERC20.sol";
```

SWLLL2-2

ACCOUNTING OPTIMISATION THROUGH UNCHECKED

SEVERITY: Informational

PATH:

src/SimpleStakingERC20.sol:deposit, withdraw (L61-74, L77-88)

REMEDIATION:

See description.

STATUS: Fixed

DESCRIPTION:

In the **deposit** and **withdraw** functions, a total supply counter **totalStakedBalance** per token is kept, as well as the user's specific balance in **stakedBalances**.

Similar to most ERC20 implementations, these additions and subtractions can be optimised by using unchecked for one, while the other will ensure there is no over-/underflow through Solidity's built-in checks.

For deposit, the **stakedBalances** operation can be placed in **unchecked**, as the **totalStakedBalance** is the sum of balances and would overflow first.

For withdraw, the **totalStakedBalance** operation can be placed in **unchecked**, as the **stakedBalances** is always less or equal to the total and so it would underflow first.

```
function deposit(ERC20 _token, uint256 _amount, address _receiver) external
nonReentrant {
    if (_amount == 0) revert AMOUNT_NULL();
    if (_receiver == address(0)) revert ADDRESS_NULL();
    if (!supportedTokens[_token].deposit) revert TOKEN_NOT_ALLOWED(_token);

    uint256 bal = _token.balanceOf(address(this));
    _token.safeTransferFrom(msg.sender, address(this), _amount);
    _amount = _token.balanceOf(address(this)) - bal; // To handle
deflationary tokens

    totalStakedBalance[_token] += _amount;
    stakedBalances[_receiver][_token] += _amount;

    emit Deposit(_token, _receiver, _amount);
}

function withdraw(ERC20 _token, uint256 _amount, address _receiver) external
nonReentrant {
    if (_amount == 0) revert AMOUNT_NULL();
    if (_receiver == address(0)) revert ADDRESS_NULL();
    if (!supportedTokens[_token].withdraw) revert TOKEN_NOT_ALLOWED(_token);

    totalStakedBalance[_token] -= _amount;
    stakedBalances[msg.sender][_token] -= _amount;

    _token.safeTransfer(_receiver, _amount);

    emit Withdraw(_token, _receiver, _amount);
}
```

Apply the unchecked scopes to the deposit and withdraw functions according to the description.

For example:

```
function deposit(ERC20 _token, uint256 _amount, address _receiver) external
nonReentrant {
    if (_amount == 0) revert AMOUNT_NULL();
    if (_receiver == address(0)) revert ADDRESS_NULL();
    if (!supportedTokens[_token].deposit) revert TOKEN_NOT_ALLOWED(_token);

    uint256 bal = _token.balanceOf(address(this));
    _token.safeTransferFrom(msg.sender, address(this), _amount);
    _amount = _token.balanceOf(address(this)) - bal; // To handle
deflationary tokens

    totalStakedBalance[_token] += _amount;
    unchecked {
        stakedBalances[_receiver][_token] += _amount;
    }

    emit Deposit(_token, _receiver, _amount);
}

function withdraw(ERC20 _token, uint256 _amount, address _receiver) external
nonReentrant {
    if (_amount == 0) revert AMOUNT_NULL();
    if (_receiver == address(0)) revert ADDRESS_NULL();
    if (!supportedTokens[_token].withdraw) revert TOKEN_NOT_ALLOWED(_token);

    stakedBalances[msg.sender][_token] -= _amount;
    unchecked {
        totalStakedBalance[_token] -= _amount;
    }

    _token.safeTransfer(_receiver, _amount);

    emit Withdraw(_token, _receiver, _amount);
}
```

SWLLL2-3

WITHDRAW MISSING INSUFFICIENT BALANCE CHECK

SEVERITY: Informational

PATH:

src/SimpleStakingERC20.sol:withdraw:L77-88

REMEDIATION:

See description.

STATUS: Fixed

DESCRIPTION:

The **withdraw** function does not have a checked for the **_amount** against the user's balance in **stakedBalances**. Instead, it relies on the subtraction to underflow if the user's balance is insufficient.

However, an underflow will revert with a generic EVM error and not tell the caller what went wrong. This hurts user experience and extensibility.

```
function withdraw(ERC20 _token, uint256 _amount, address _receiver) external
nonReentrant {
    if (_amount == 0) revert AMOUNT_NULL();
    if (_receiver == address(0)) revert ADDRESS_NULL();
    if (!supportedTokens[_token].withdraw) revert TOKEN_NOT_ALLOWED(_token);

    totalStakedBalance[_token] -= _amount;
    stakedBalances[msg.sender][_token] -= _amount;

    _token.safeTransfer(_receiver, _amount);

    emit Withdraw(_token, _receiver, _amount);
}
```

A custom error should be added that is reverted upon when the user's balance is insufficient. As a result, the subtraction can also be placed in unchecked (and together with the remediation of issue 2, both can be placed in unchecked):

```
function withdraw(ERC20 _token, uint256 _amount, address _receiver) external
nonReentrant {
    if (_amount == 0) revert AMOUNT_NULL();
    if (stakedBalances[msg.sender][_token] < _amount) revert
INSUFFICIENT_BALANCE();
    if (_receiver == address(0)) revert ADDRESS_NULL();
    if (!supportedTokens[_token].withdraw) revert TOKEN_NOT_ALLOWED(_token);

    unchecked {
        totalStakedBalance[_token] -= _amount;
        stakedBalances[msg.sender][_token] -= _amount;
    }

    _token.safeTransfer(_receiver, _amount);

    emit Withdraw(_token, _receiver, _amount);
}
```

SWLLL2-5

MISSING INPUT VALIDATION IN ZAP CONTRACT

SEVERITY: Informational

PATH:

src/Zap.sol::ethZapIn():L34-L40

src/Zap.sol::stETHZapIn():L42-L50

src/Zap.sol::eETHZapIn():L52-L60

REMEDIATION:

See description.

STATUS: Fixed

DESCRIPTION:

The **Zap.sol** contract includes three functions (**ethZapIn()**, **stETHZapIn()**, and **eETHZapIn()**) for depositing assets into a staking contract. However, none of the three functions include a check to ensure that the amount of asset being deposited is greater than zero.

Although the staking contract will ultimately revert if zero assets are attempted to be staked, the absence of input validation checks in the **Zap.sol** contract could prolong the process unnecessarily. By incorporating these checks directly into the **Zap.sol** contract, transactions with zero assets could be rejected immediately, enhancing efficiency and user experience.

It's worth noting that the **Zap.sol** contract already imports **ISimpleStakingERC20.sol**, which contains the necessary error definitions. Therefore, there's no need to declare new errors; simply adding input validation checks suffices.

```
function ethZapIn() external payable {
    // Wrap ETH to wETH
    weth.deposit{value: msg.value}();

    // Deposit wETH to staking contract
    stakingContract.deposit(weth, msg.value, msg.sender);
}

function stETHZapIn(uint256 _amount) external {
    // Transfer stETH from msg.sender to this contract, sometimes 1 or 2 wei
can be missing.
    uint256 bal = stETH.balanceOf(address(this));
    stETH.transferFrom(msg.sender, address(this), _amount);
    bal = stETH.balanceOf(address(this)) - bal;

    // Deposit wstETH to staking contract
    stakingContract.deposit(ERC20(address(wstETH)), wstETH.wrap(bal),
msg.sender);
}

function eETHZapIn(uint256 _amount) external {
    // Transfer eETH from msg.sender to this contract, sometimes 1 or 2 wei
can be missing.
    uint256 bal = eETH.balanceOf(address(this));
    eETH.transferFrom(msg.sender, address(this), _amount);
    bal = eETH.balanceOf(address(this)) - bal;

    // Deposit eETH to staking contract
    stakingContract.deposit(ERC20(address(weETH)), weETH.wrap(bal),
msg.sender);
}
```

Implement input validation checks at the beginning of each function to verify that _amount is greater than zero, for example:

```
    function ethZapIn() external payable {
+       if (msg.value == 0) revert AMOUNT_NULL();
        // Wrap ETH to wETH
        weth.deposit{value: msg.value}();

        // Deposit wETH to staking contract
        stakingContract.deposit(weth, msg.value, msg.sender);
    }
```


hexens
×
Swell