Swell Audit Report

Prepared by Cyfrin

Version 2.0

Lead Auditors

Dacian

Carlitox477

February 22, 2024

Contents

1 About Cyfrin

Cyfrin is a Web3 security company dedicated to bringing industry-leading protection and education to our partners and their projects. Our goal is to create a safe, reliable, and transparent environment for everyone in Web3 and DeFi. Learn more about us at cyfrin.io.

2 Disclaimer

The Cyfrin team makes every effort to find as many vulnerabilities in the code as possible in the given time but holds no responsibility for the findings in this document. A security audit by the team does not endorse the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the solidity implementation of the contracts.

3 Risk Classification

	Impact: High	**Impact: Medium**	**Impact: Low**
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

4 Protocol Summary

Swell is a Liquid Staking Derivatives (LSD) protocol which allows users to participate in and benefit from Ethereum staking without having their funds locked up or running validators. Users are able to deposit their `ETH` into the protocol in exchange for `swETH`, Swell's liquid ERC20 token.

Once enough `ETH` has been deposited into the protocol, Swell allows trusted operators to utilize this `ETH` to run validators and participate in Ethereum's proof of stake consensus layer, thereby earning rewards. As rewards are earned Swell re-calculates the `swETH/ETH` exchange rate such that holders of `swETH` can burn their `swETH` to withdraw a greater amount of `ETH` than they originally deposited due to the updated exhange rate reflecting the rewards earned by Swell's validators. Operators are compensated with a percentage of the total `ETH` rewards in `swETH`, based on their proportion of active validators. This percentage is defined by the Swell protocol. Additionally, the Swell protocol also receives a proportion of these rewards in `swETH`.

If the `swETH/ETH` exchange rate would decrease due to validator slashing penalties or any other reason, Swell initiates a lockdown which pauses all key protocol functions allowing the project team to investigate and unpause functionality as they deem it safe to do so.

A high amount of centralization is part of Swell's design; Swell is a highly-permissioned protocol with multiple permissioned actors who have the ability to call functions which:

- Pause/unpause the protocol including pausing withdrawals
- Enable/disable operators and delete validators
- Change the `swETH/ETH` exchange rate

All major smart contracts are also upgradeable meaning their implementation can be changed at any time. While this highly-permissioned design serves to significantly simplify the protocol helping to reduce the attack surface from non-permissioned actors, it also requires users who engage with the protocol to put a high degree of trust in the Swell team.

Validator states:

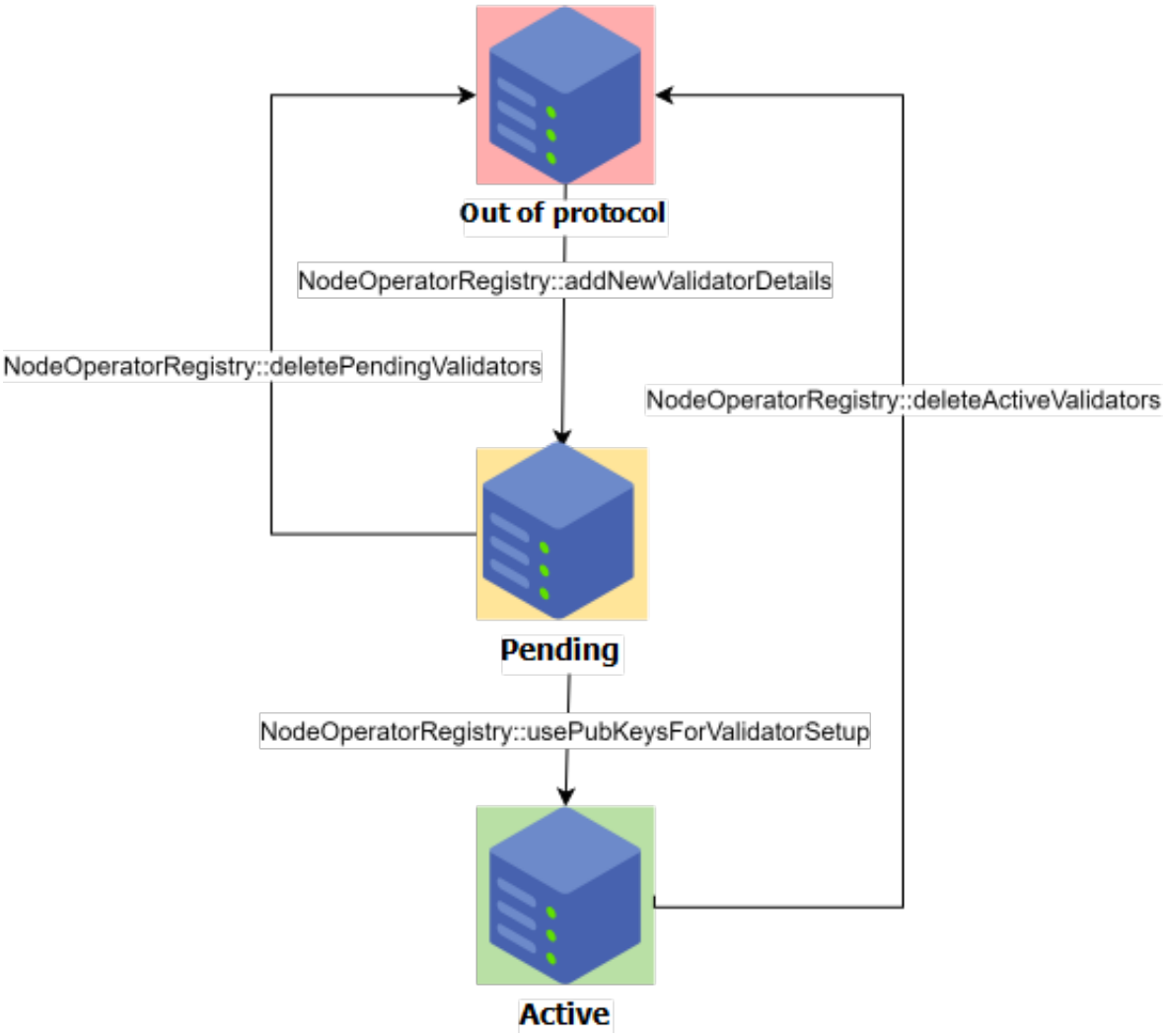

Figure 1: Validator states

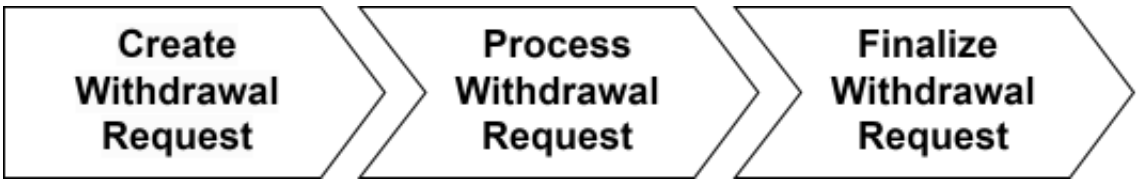

Figure 2: Withdrawal process

Repricing mechanism: Repricing mechanism consist in updating `swETH/ETH` due to ETH rewards, and minting swETH to compensate Swell protocol and operators for running active validators. To do this the SwellLib.BOT actor submit a snapshot of: * $Deposited\ ETH$: `ETH` deposited in `Deposit contract`. * $Pending\ ETH$: `ETH` deposited in Ethereum 1 network that is accepted in Ethereum 2 network but it has not been used to attest and propose blocks yet. * $Activated\ ETH$: `ETH` in Ethereum 2 network that is being used to attest and propose blocks. This value includes $Exiting\ ETH$ * $Exiting\ ETH$: `ETH` that is in the process of halting attesting to and proposing blocks (still attesting to and proposing blocks) * $ETH\ Rewards$: `ETH` that has been obtained as rewards since last reserves update.

Then, the protocol compares most of these values with Chainlink PoR oracle to validate them or revert the transaction. To avoid possible front-running Swell protocol is also forced to send $swETH\ current\ supply_{informed}$ to perform next calculations:

$$Pre\ Reward\ ETH\ =\ Deposited\ ETH\ +\ Pending\ ETH\ +\ Activated\ ETH\ -\ Exiting\ ETH$$

$$Total\ Reserves\ =\ Pre\ Reward\ ETH\ + ETH\ Rewards$$

$$swETH\ to\ mint = swETH\ current\ supply_{informed} \times \frac{ETH\ Rewards \times Fee_{operator\ +\ treasury}\%}{Total\ Reserves\ -\ ETH\ Rewards \times Fee_{operator\ +\ treasury}\%}$$

$$swETH/ETH_{new} = \frac{Total\ Reserves}{swETH\ current\ supply_{informed} + swETH\ to\ mint}$$

swETH withdrawal process:

The `swETH` withdrawal process is handled by the `swEXIT` contract, an ERC-721 token that can be minted by calling `createWithdrawRequest` and specifying the amount of `swETH` to burn in exchange for `ETH` once the withdrawal is processed. Although anyone can mint `swETH`, withdrawal requests can only be created by whitelisted addresses when the whitelisting mechanism is enabled.

Withdrawal requests are processed by a bot when `processWithdrawals` is called. Given that the `swETH/ETH` rate can change between the creation and the processing of a withdrawal, the minimum rate at these two moments is used to process the withdrawal.

After a withdrawal request has been processed the ETH can be claimed by calling `finalizeWithdrawal`. Once claimed, the NFT representing the processed withdrawal request is burned and the ETH is sent to the owner of the NFT.

$$ETHswETHRate\ to\ use(wr) = min(wr.ETHSwETHRate_{creation}, wr.ETHSwETHRate_{processing})$$
$$ETH\ to\ receive(wr) = wr.swETHAmount \times ETHswETHRate\ to\ use(wr)$$

where wr represents a withdrawal request.

5 Audit Scope

The Barracuda Upgrade is a significant upgrade to Swell's protocol which aims to:

- allow holders of `swETH` to withdraw by burning their tokens in exchange for `ETH`
- strengthen the security of the protocol through implementing more granular permissioned roles and a lock-down mechanism

The following contracts were included in the scope for this audit:

```
contracts/implementations/AccessControlManager.sol
contracts/implementations/DepositManager.sol
contracts/implementations/NodeOperatorRegistry.sol
contracts/implementations/RepricingOracle.sol
contracts/implementations/swETH.sol
contracts/implementations/swEXIT.sol
contracts/implementations/Whitelist.sol
contracts/interfaces/IAccessControlManager.sol
contracts/interfaces/IDepositManager.sol
contracts/interfaces/INodeOperatorRegistry.sol
contracts/interfaces/IRepricingOracle.sol
contracts/interfaces/IswETH.sol
contracts/interfaces/IswEXIT.sol
contracts/interfaces/IWhitelist.sol
contracts/libraries/DepositDataRoot.sol
contracts/libraries/EnumberableSetValidatorDetails.sol
contracts/libraries/Repricing.sol
contracts/libraries/SwellLib.sol
contracts/vendors/AggregatorV3Interface.sol
contracts/vendors/IDepositContract.sol
contracts/vendors/IPorAddresses.sol
contracts/vendors/IRateProvider.sol
```

6 Executive Summary

Over the course of 18 days, the Cyfrin team conducted an audit on the Swell smart contracts provided by Swell. In this period, a total of 24 issues were found.

The findings consist of 2 Medium & 11 Low severity issues with the remainder being informational and gas optimizations. Both Medium findings related to the refactored permission structure which allowed the `BOT` actor to:

- delete active validators when `BOT` actions were paused
- subtly rug-pull withdrawals by processing them with an arbitrary exchange rate

The 11 Low findings included a mix of issues such as:

- unnecessary precision loss when minting `swETH` and calculating validator rewards (not introduced in the proposed upgrade but present on mainnet)
- corruption of `NodeRegistryManager` storage when updating operator address (not introduced in the proposed upgrade but present on mainnet)
- not checking for stale data when using Swell's Chainlink Proof of Reserves oracle
- unexpected behavior and invalid states (1 present on mainnet)

In total 4 Low findings were not introduced in the proposed upgrade but are present on mainnet.

As part of our testing we developed a custom invariant fuzzer using Echidna and Medusa which found a number of Low issues related to invalid states; the invariant fuzzing code has been delivered to Swell as an extra deliverable at the completion of the audit.

Summary

Project Name	Swell
Repository	v3-contracts-lst
Commit	a95ea7942ba8...
Audit Timeline	Jan 29th - Feb 21st
Methods	Manual Review, Stateful Fuzzing

Issues Found

Critical Risk	0
High Risk	0
Medium Risk	2
Low Risk	11
Informational	4
Gas Optimizations	7
Total Issues	24

Summary of Findings

[M-1] `SwellLib.BOT` can delete active validators when bot methods are paused	Resolved
[M-2] `SwellLib.BOT` can subtly rug-pull withdrawals by setting `_processedRate = 0` when calling `swEXIT::processWithdrawals`	Resolved
[L-01] Precision loss in `swETH::reprice` from unnecessary division before multiplication	Acknowledged
[L-02] `swEXIT::setWithdrawRequestMaximum` and `setWithdrawRequestMinimum` lacking validation can lead to a state where `withdrawRequestMinimum > withdrawRequestMaximum`	Resolved
[L-03] `swExit::getProcessedRateForTokenId` returns `true` with valid `processedRate` for non-existent `tokenId` input	Resolved
[L-04] Check for staleness of data when fetching Proof of Reserves via Chainlink `Swell ETH PoR` Oracle	Resolved
[L-05] `swETH::reprice` may run out of gas or become exorbitantly expensive when scaling to large number of validator operators due to iterating over them all	Acknowledged

[L-06] `NodeOperatorRegistry::updateOperatorControllingAddress` allows to override `_newOperatorAddress` if its address is already assigned to an operator ID	Resolved
[L-07] Allowing anyone to finalize any withdrawal can lead to integration problems for smart contract allowed to receive ETH	Resolved
[L-08] Multiple attack paths to force `swETH::reprice` to revert by increasing or decreasing swETH total supply	Resolved
[L-09] Rewards unable to be distributed when all active validators are deleted during repricing	Resolved
[L-10] Repricing with small rewards results in an invalid state where `ETH` reserves increase, `swETH to ETH` exchange rate increases, but no rewards are paid out to operators or treasury	Acknowledged
[L-11] Precision loss in `swETH::_deposit` from unnecessary hidden division before multiplication	Resolved
[I-1] Emit `ETHSent` event when sending eth	Resolved
[I-2] Use Checks-Effects-Interactions pattern in `swEXIT::createWithdrawRequest`	Resolved
[I-3] Missing events in `NodeOperatorRegistry` update methods	Resolved
[I-4] Refactor identical code in `NodeOperatorRegistry::getNextValidatorDetails`	Resolved
[G-1] Cache storage variables in memory when read multiple times without being changed	Resolved
[G-2] Cache array length outside of loops and consider unchecked loop incrementing	Resolved
[G-3] `NodeOperatorRegistry::_parsePubKeyToString`: Use shift operations rather than division/multiplication when dividend/factor is a power of 2	Resolved
[G-4] Use `totalReserves - rewardsInETH.unwrap()` rather than `_preRewardETHReserves - rewardsInETH.unwrap() + _newETHRewards` in `swETH::reprice`	Resolved
[G-5] Remove redundant pause checks	Resolved
[G-6] Refactor `RepricingOracle::handleReprice`, `_assertRepricingSnapshotValidity` and `_repricingPeriodDeltas`	Acknowledged
[G-7] Use constant for unchanging deposit amount	Acknowledged

7 Findings

7.1 Medium Risk

7.1.1 `SwellLib.BOT` can delete active validators when bot methods are paused

Description: Almost all of the functions callable by `SwellLib.BOT` contain the following check to prevent bot functions from working when bot methods are paused:

```
if (AccessControlManager.botMethodsPaused()) {
  revert SwellLib.BotMethodsPaused();
}
```

The one exception is `NodeOperatorRegistry::deleteActiveValidators` which is callable by `SwellLib.BOT` even when bot methods are paused. Consider:

- adding a similar check to this function such that `SwellLib.BOT` is not able to call it when bot methods are paused
- alternatively add an explicit comment to this function stating that it should be callable by `SwellLib.BOT` even when bot methods are paused.

One possible implementation for the first solution:

```
bool isBot = AccessControlManager.hasRole(SwellLib.BOT, msg.sender);

// prevent bot from calling this function when bot methods are paused
if(isBot && AccessControlManager.botMethodsPaused()) {
  revert SwellLib.BotMethodsPaused();
}

// function only callable by admin & bot
if (!AccessControlManager.hasRole(SwellLib.PLATFORM_ADMIN, msg.sender) && !isBot) {
  revert OnlyPlatformAdminOrBotCanDeleteActiveValidators();
}
```

Swell: Fixed in commit 1a105b7.

Cyfrin: Verified.

7.1.2 `SwellLib.BOT` can subtly rug-pull withdrawals by setting `_processedRate = 0` when calling `swEXIT::processWithdrawals`

Description: When users create a withdrawal request, their `swETH` is burned then the current exchange rate `rateWhenCreated` is fetched from `swETH::swETHToETHRate`:

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

However `SwellLib.BOT` can pass an arbitrary value for `_processedRate` when calling `swEXIT::processWithdrawals`:

```
function processWithdrawals(
  uint256 _lastTokenIdToProcess,
  uint256 _processedRate
) external override checkRole(SwellLib.BOT) {
```

The final rate used is the lesser of `rateWhenCreated` and `_processedRate`:

```
uint256 finalRate = _processedRate > rateWhenCreated
  ? rateWhenCreated
  : _processedRate;
```

This final rate is multiplied by the requested withdrawal amount to determine the actual amount sent to the user requesting a withdrawal:

```
uint256 requestExitedETH = wrap(amount).mul(wrap(finalRate)).unwrap();
```

Hence `SwellLib.BOT` can subtly rug-pull all withdrawals by setting `_processedRate = 0` when calling `swEXIT::processWithdrawals`.

Recommended Mitigation: Two possible mitigations:

1) Change `swEXIT::processWithdrawals` to always fetch the current rate from `swETH::swETHToETHRate`
2) Only allow `swEXIT::processWithdrawals` to be called by the `RepricingOracle` contract which calls it correctly.

Swell: Fixed in commits c6f8708, 64cfbdb.

Cyfrin: Verified.

7.2 Low Risk

7.2.1 Precision loss in `swETH::reprice` from unnecessary division before multiplication

Description: `swETH::reprice` L281-286 performs unnecessary division before multiplication when calculating node operator rewards which negatively impacts node operator rewards due to precision loss:

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

nodeOperatorRewards = nodeOperatorRewardPortion
  .mul(rewardsInSwETH) // @audit mult after division
  .unwrap();
```

Refactor to perform division after multiplication:

```
nodeOperatorRewards = wrap(nodeOperatorRewardPercentage)
  .mul(rewardsInSwETH)
  .div(wrap(rewardPercentageTotal))
  .unwrap();
```

A similar issue occurs when calculating operators reward share L310-313:

```
uint256 operatorsRewardShare = wrap(operatorActiveValidators)
  .div(totalActiveValidators)
  .mul(wrap(nodeOperatorRewards)) // @audit mult after division
  .unwrap();
```

This can be similarly refactored to prevent the precision loss by performing multiplication first:

```
uint256 operatorsRewardShare = wrap(operatorActiveValidators)
  .mul(wrap(nodeOperatorRewards))
  .div(totalActiveValidators)
  .unwrap();
```

This issue has not been introduced in the new changes but is in the mainnet code (1, 2).

There is still one potential precision loss remaining as `rewardsInSwETH` which has had a division performed then gets multiplied but attempting to refactor this out resulted in a "stack too deep" error so it may be unavoidable.

Swell: Acknowledged.

7.2.2 `swEXIT::setWithdrawRequestMaximum` and `setWithdrawRequestMinimum` lacking validation can lead to a state where `withdrawRequestMinimum > withdrawRequestMaximum`

Description: Invariant `withdrawRequestMinimum <= withdrawRequestMaximum` must always hold, however this is not checked when new min/max withdraw values are set. Hence it is possible to enter a non-sensical state where `withdrawRequestMinimum > withdrawRequestMaximum`.

Recommended mitigation:

```
  function setWithdrawRequestMaximum(
    uint256 _withdrawRequestMaximum
  ) external override checkRole(SwellLib.PLATFORM_ADMIN) {
+   require(withdrawRequestMinimum <= _withdrawRequestMaximum);

    emit WithdrawalRequestMaximumUpdated(
      withdrawRequestMaximum,
      _withdrawRequestMaximum
    );
    withdrawRequestMaximum = _withdrawRequestMaximum;
  }

  function setWithdrawRequestMinimum(
    uint256 _withdrawRequestMinimum
  ) external override checkRole(SwellLib.PLATFORM_ADMIN) {
+   require(_withdrawRequestMinimum <= withdrawRequestMaximum);

    emit WithdrawalRequestMinimumUpdated(
      withdrawRequestMinimum,
      _withdrawRequestMinimum
    );
    withdrawRequestMinimum = _withdrawRequestMinimum;
  }
```

Swell: Fixed in commit a9dfe5c.

Cyfrin: Verified.

7.2.3 `swExit::getProcessedRateForTokenId` returns `true` with valid `processedRate` for non-existent `tokenId` input

Description: `swExit::getProcessedRateForTokenId` returns `true` with valid `processedRate` for non-existent `tokenId` input.

Impact: This `public` function can return valid output for invalid input. Currently it only appears to be used by `finalizeWithdrawal` where this behavior does not seem to be further exploitable as that function checks for non-existent tokens before calling `getProcessedRateForTokenId`.

Proof of Concept: Add the following PoC to `getProcessedRateForTokenId.test.ts`:

```
  it("Should return false for isProcessed when tokens have been processed but this token doesn't
  ↪  exist", async () => {
    await createWithdrawRequests(Deployer, 5);

    await swEXIT_Deployer.processWithdrawals(4, parseEther("1"));

    // @audit this test fails
    expect(await getProcessedRateForTokenId(0)).eql({
      isProcessed: false,               // @audit returns true
      processedRate: BigNumber.from(0), // @audit returns > 0
    });
  });
```

Recommended Mitigation: `swExit::getProcessedRateForTokenId` should `return(false, 0)` when `tokenId` doesn't exist. It appears that the only edge case which is currently unhandled by this function is when `tokenId = 0`.

Swell: Fixed in commits 4c8cbfd, 262db73.

Cyfrin: Verified.

7.2.4 Check for staleness of data when fetching Proof of Reserves via Chainlink `Swell ETH PoR` Oracle

Description: `RepricingOracle::_assertRepricingSnapshotValidity` uses the `Swell ETH PoR` Chainlink Proof Of Reserves Oracle to fetch an off-chain data source for Swell's current reserves.

The Oracle `Swell ETH PoR` is listed on Chainlink's website as having a heartbeat of `86400` seconds (check the "Show More Details" box in the top-right corner of the table), however no staleness check is implemented by `RepricingOracle`:

```
// @audit no staleness check
(, int256 externallyReportedV3Balance, , , ) = AggregatorV3Interface(
  ExternalV3ReservesPoROracle
).latestRoundData();
```

Impact: If the `Swell ETH PoR` Chainlink Proof Of Reserves Oracle has stopped functioning correctly, `RepricingOracle::_assertRepricingSnapshotValidity` will continue processing with stale reserve data as if it were fresh.

Recommended Mitigation: Implement a staleness check and if the Oracle is stale, either revert or skip using it as the code currently does if the oracle is not set.

For multi-chain deployments ensure that a correct staleness check is used for each feed as the same feed can have different heartbeats on different chains.

Consider adding an off-chain bot that periodically checks if the Oracle has become stale and if it has, raises an internal alert for the team to investigate.

Swell: Fixed in commit 84a6517.

Cyfrin: Verified.

7.2.5 `swETH::reprice` may run out of gas or become exorbitantly expensive when scaling to large number of validator operators due to iterating over them all

Description: `swETH::reprice` loops through all validator operators to pay out their share of rewards:

```
// @audit may run out of gas for larger number of validator operators
// or make repricing exorbitantly expensive
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;
  }
}
```

If Swell scales to a large number of validators `swETH::reprice` may revert due to out of gas or make the reprice operation exorbitantly expensive. `NodeOperatorRegistry::getNextValidatorDetails` may be similarly affected.

Currently this represents a low risk for Swell as the protocol uses a small set of "permissioned group of professional node operators".

However Swell intends to transition away from this: *"The subsequent iterations will see the operator set **expand** and ultimately be permissionless.."*

As Swell expands the operator set this issue will become a more serious concern and may require mitigation.

Swell: Acknowledged.

7.2.6 `NodeOperatorRegistry::updateOperatorControllingAddress` allows to override `_newOperatorAddress` if its address is already assigned to an operator ID

Description: Current implementation does not check if the new assigned address has already been assigned to an operator ID. As a consequence, its current value can be over written in mapping `getOperatorIdForAddress`, and `getOperatorForOperatorId` will have 2 operator IDs pointing to the same operator.

The direct consequences of this are on `_getOperatorSafe` and `_getOperatorIdSafe`, which will only return data for the new assigned operator ID.

Therefore:

- `NodeOperatorRegistry::getOperatorsPendingValidatorDetails` won't be able to return old `_newOperatorAddress` associated validators details
- `NodeOperatorRegistry::getOperatorsActiveValidatorDetails` won't be able to return old `_newOperatorAddress` associated active validators details
- `enableOperator` won't be able to enable old operator record
- `disableOperator` **won't be able to disable old operator record**. This can affect function `usePubKeysForValidatorSetup` given that the protocol won't be able to disable already enabled public key to be used for validator setup given that there is no way to modify previous `getOperatorForOperatorId[_newOperatorAddress].enabled` storage and force the function to revert. Given that the only one allowed to call the function is the BOT by previously calling `DepositManager::setupValidators` the impact is limited.
- `updateOperatorRewardAddress` won't be able to modify reward address from old operator record
- `updateOperatorName` won't be able to modify name from old operator record

This issue has not been introduced in the new changes but is in the mainnet code.

Proof Of Concept: Add the following test to `updateOperatorFields.test.ts`:

```
it("Should revert updating operator controlling address to existing address", async () => {
  // create another operator
  await NodeOperatorRegistry_Deployer.addOperator(
    "OPERATOR_2",
    NewOperator.address,
    NewOperator.address
  );

  // attempt to update first operator's controlling address to be
  // the same as the newly created operator - should revert but doesn't
  await NodeOperatorRegistry_Deployer.updateOperatorControllingAddress(
    Operator.address,
    NewOperator.address
  );
});
```

Recommended mitigation: Check that `_newOperatorAddress` is not already assigned to an operator (similar to `addOperator` which already does this, may wish to create a new private or public function for code reuse):

```
  function updateOperatorControllingAddress(
    address _operatorAddress,
    address _newOperatorAddress
  )
    external
    override
    checkRole(SwellLib.PLATFORM_ADMIN)
    checkZeroAddress(_newOperatorAddress)
  {

    if (_operatorAddress == _newOperatorAddress) {
        revert CannotSetOperatorControllingAddressToSameAddress();
    }
+   if(getOperatorIdForAddress[_newOperatorAddress] != 0){
+       revert CannotUpdateOperatorControllingAddressToAlreadyAssignedAddress();
+   }

    uint128 operatorId = _getOperatorIdSafe(_operatorAddress);

    getOperatorIdForAddress[_newOperatorAddress] = operatorId;
    getOperatorForOperatorId[operatorId]
      .controllingAddress = _newOperatorAddress;

    delete getOperatorIdForAddress[_operatorAddress];
  }
```

Swell: Fixed in commit 55c7d5f.

Cyfrin: Verified.

7.2.7 Allowing anyone to finalize any withdrawal can lead to integration problems for smart contract allowed to receive ETH

Description: Current implementation of `swEXIT::finalizeWithdrawal` allows anyone to finalize any withdrawal request which is already processed. However this design decision make the strong assumption that an NFT owner always wants to finalize a withdrawal, which might not be always the case.

Impact: Allowing anyone to finalize any withdrawal request already processed can lead to stuck ETH in some smart contracts

POC: Assume a protocol which goals is facilitating NFT auctions, with auctions that can accept any token or ETH. Bidders has a record for the amount of tokens/ETH they are offering for an NFT, so the smart contract implement a `receive` function to accept ETH.

Eve initiate a withdrawal request, but given that she urge for ETH she decide to use this protocol to sell her NFT in an auction. To do this, she must transfer the NFT to the auction contract.

Alice decide to bid for the NFT, and at the end of the auction she wins, now she has to claim the NFT (the auction contract is the owner of the NFT right now).

The swEXIT NFT is processed before Alice intend to claim it, Eve calls `finalizeWithdrawal` with the NFT in the auction contract, given that this contract is allowed to receive ETH and it is the NFT owner the transaction does not revert, and the ETH associated to the NFT now is stuck forever in the auction contract, Alice cannot claim nothing now.

Recommended Mitigation: Only allowed the owner of the NFT to finalize a withdrawal

```
    function finalizeWithdrawal(uint256 tokenId) external override {
        if (AccessControlManager.withdrawalsPaused()) {
        revert WithdrawalsPaused();
        }

        address owner = _ownerOf(tokenId);

-       if (owner == address(0)) {
-           revert WithdrawalRequestDoesNotExist();
+       if (owner == msg.sender) {
+           revert WithdrawalRequestFinalizationOnlyAllowedForNFTOwner();
        }
```

Swell: Fixed in commit b5d7a19.

Cyfrin: Verified.

7.2.8 Multiple attack paths to force `swETH::reprice` to revert by increasing or decreasing swETH total supply

Description: The current total swETH supply is used in `swETH::reprice` to enforce the maximum allowed total swETH supply difference during repricings. Total supply can decrease for 2 reasons:

1. Withdrawal being finalized
2. User calls `swETH::burn` to burn their own swETH

Total supply can also increase by users calling `swETH::deposit`.

The closer the current supply difference is to the maximum tolerated difference percentage, the greater chance an attacker can front-run the repricing transaction causing it to revert by:

1. Depositing a large enough amount of ETH via `swETH::deposit` to increase total supply
2. Burning their own swETH to decrease total supply
3. Finalizing one or more withdrawals (users can finalize others withdrawals) to decrease total supply

Recommended mitigation: Some possible mitigations include:

- Add a burner role and assigned it only to `swEXIT`, also add the corresponding modifier to check this role to `swETH::burn`
- Only allow the owner of an NFT to finalize their owned withdrawal requests

However these potential mitigations restrict functionality while still enabling an attacker to revert the reprice via the `swETH::deposit` route. Another option would be to have the bot perform the repricing transaction through a service such as flashbots such that the transaction can't be front-run; this would prevent all of the attack paths while still preserving the ability for users to burn their swETH and to finalize others withdrawals.

Swell: Using flashbots to perform repricing transactions.

7.2.9 Rewards unable to be distributed when all active validators are deleted during repricing

Description: Invariant fuzzing found an interesting edge-case during repricing if:

1) there are rewards to distribute which were accrued in the last period,
2) all the current active validators are being deleted in the repricing operation

Because the validators are deleted first the reprice transaction reverts with `NoActiveValidators` error.

No repricings will be possible until new active validators are added, and when that occurs the new validators will receive the rewards that were generated by the old validators which were deleted. Additionally Aaron confirmed

on TG: *it is theoretically possible for fees to be generated without any active validators as any ETH sent to the `DepositManager` is considered rewards and eligible for fees.*

Recommended Mitigation: During repricing if there are no active validators but rewards to be distributed, instead of reverting the rewards should go to the Swell treasury.

Swell: Fixed in commit 5594e20.

Cyfrin: Verified.

7.2.10 Repricing with small rewards results in an invalid state where `ETH` reserves increase, `swETH to ETH` exchange rate increases, but no rewards are paid out to operators or treasury

Description: Invariant fuzzing used repricings with small rewards to reach an invalid state where `ETH` reserves increase, `swETH : ETH` exchange rate increases, but no rewards are paid out to operators or treasury.

Proof of Concept: During repricing:

1) there is no minimum value enforced by either `RepricingOracle` for `_snapshot.rewardsPayableForFees` or `swETH::reprice` for `_newETHRewards`
2) in `swETH::reprice` there is no check for rounding down to zero precision loss when calculating `rewardsInSwETH`

This results in the fuzzer reaching an invalid state where:

1) by calling `RepricingOracle::submitSnapshotV2` with small values for `_snapshot.rewardsPayableForFees`, this results in `swETH::reprice` being called with small `_newETHRewards`
2) inside `swETH::reprice` the small `_newETHRewards` triggers a rounding down to zero precision loss in the rewards calculation of `rewardsInSwETH` so rewards are never distributed
3) however `swETH::reprice` does update `lastRepriceETHReserves` using the small positive `_newETHRewards` value and the transaction completes successfully.

This results in an invalid state where:

1) `swETH::lastRepriceETHReserves` increases
2) `swETH : ETH` exchange rate increases
3) no rewards are being paid out to operators/treasury

This simplified PoC can be added to `reprice.test.ts`:

```
  it("audit small rewards not distributed while reserves and exchange rate increasing", async () => {
    const swellTreasuryRewardPercentage = parseEther("0.1");

    await swETH_Deployer.setSwellTreasuryRewardPercentage(
      swellTreasuryRewardPercentage
    );

    await swETH_Deployer.deposit({
      value: parseEther("1000"),
    });
    const preRewardETHReserves = parseEther("1100");

    const swETHSupply = parseEther("1000");

    const ethRewards = parseUnits("1", "wei");

    const swellTreasuryPre = await swETH_Deployer.balanceOf(SwellTreasury.address);
    const ethReservesPre = await swETH_Deployer.lastRepriceETHReserves();
    const rateBefore = await swETH_Deployer.swETHToETHRate();

    swETH_Bot.reprice(
        preRewardETHReserves,
        ethRewards,
        swETH_Deployer.totalSupply());

    const swellTreasuryPost = await swETH_Deployer.balanceOf(SwellTreasury.address);
    const ethReservesPost = await swETH_Deployer.lastRepriceETHReserves();
    const rateAfter = await swETH_Deployer.swETHToETHRate();

    // no rewards distributed to treasury
    expect(swellTreasuryPre).eq(swellTreasuryPost);

    // exchange rate increases
    expect(rateBefore).lt(rateAfter);

    // reserves increase
    expect(ethReservesPre).lt(ethReservesPost);

    // repricing using small `_newETHRewards` can lead to increasing reserves
    // and increasing exchange rate without reward payouts
  });
```

This was not introduced in the new changes but is present in the current mainnet code [1, 2].

Swell: Acknowledged.

7.2.11 Precision loss in `swETH::_deposit` from unnecessary hidden division before multiplication

Description: `swETH::_deposit` L170 contains a hidden unnecessary division before multiplication as the call to `_ethToSwETHRate` performs a division which then gets multiplied by `msg.value`:

```
uint256 swETHAmount = wrap(msg.value).mul(_ethToSwETHRate()).unwrap();
// @audit expanding this out
// wrap(msg.value).mul(_ethToSwETHRate()).unwrap();
// wrap(msg.value).mul(wrap(1 ether).div(_swETHToETHRate())).unwrap();
```

This issue has not been introduced in the new changes but is in the mainnet code.

Impact: Slightly less `swETH` will be minted to depositors. While the amount by which individual depositors are short-changed is individually small, the effect is cumulative and increases as depositors and deposit size increase.

Proof of Concept: This stand-alone stateless fuzz test can be run inside Foundry to prove this as well as provided hard-coded test cases:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.23;

import {UD60x18, wrap} from "@prb/math/src/UD60x18.sol";

import "forge-std/Test.sol";

// run from base project directory with:
// (fuzz test) forge test --match-test FuzzMint -vvv
// (hardcoded) forge test --match-test HardcodedMint -vvv
contract MintTest is Test {

    uint256 private constant SWETH_ETH_RATE = 1050754209601187151; //as of 2024-02-15

    function _mintOriginal(uint256 inputAmount) private pure returns(uint256) {
        // hidden division before multiplication
        // wrap(inputAmount).mul(_ethToSwETHRate()).unwrap();
        // wrap(inputAmount).mul(wrap(1 ether).div(_swETHToETHRate())).unwrap()

        return wrap(inputAmount).mul(wrap(1 ether).div(wrap(SWETH_ETH_RATE))).unwrap();
    }

    function _mintFixed(uint256 inputAmount) private pure returns(uint256) {
        // refactor to perform multiplication before division
        // wrap(inputAmount).mul(wrap(1 ether)).div(_swETHToETHRate()).unwrap();

        return wrap(inputAmount).mul(wrap(1 ether)).div(wrap(SWETH_ETH_RATE)).unwrap();
    }

    function test_FuzzMint(uint256 inputAmount) public pure {
        uint256 resultOriginal = _mintOriginal(inputAmount);
        uint256 resultFixed    = _mintFixed(inputAmount);

        assert(resultOriginal == resultFixed);
    }

    function test_HardcodedMint() public {
        // found by fuzzer
        console.log(_mintFixed(3656923177187149889) - _mintOriginal(3656923177187149889)); // 1

        // 100 eth
        console.log(_mintFixed(100 ether) - _mintOriginal(100 ether)); // 21

        // 1000 eth
        console.log(_mintFixed(1000 ether) - _mintOriginal(1000 ether)); // 215

        // 10000 eth
        console.log(_mintFixed(10000 ether) - _mintOriginal(10000 ether)); // 2159
    }
}
```

Recommended Mitigation: Refactor to perform multiplication before division:

```
uint256 swETHAmount = wrap(msg.value).mul(wrap(1 ether)).div(_swETHToETHRate()).unwrap();
```

Swell: Fixed in commit cb093ea.

Cyfrin: Verified.

7.3 Informational

7.3.1 Emit `ETHSent` event when sending eth

Description: `DepositManager::receive` emits an `ETHReceived` event when receiving eth, but `transferETHForWithdrawRequests` does not emit any events when sending eth; consider also emitting an `ETHSent` event when sending eth.

Swell: Fixed in commit c82dd3c.

Cyfrin: Verified.

7.3.2 Use Checks-Effects-Interactions pattern in `swEXIT::createWithdrawRequest`

Description: The current implementation uses `_safeMint` before modifying state variables:

- `withdrawalRequests[tokenId]`
- `exitingETH`
- `_lastTokenIdCreated`

This allows possible re-entrancy where the receiver and access the non-updated state variables. While during the audit no meaningful permissionless attack vectors related to this issue were found, to follow best security practices it is advisable to move `_safeMint` to the end of the function:

```
  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
+   uint256 tokenId = ++_lastTokenIdCreated; // Starts 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;
+   _safeMint(msg.sender, tokenId);

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

  }
```

Swell: Fixed in commits d13aa43, 3f85df3.

Cyfrin: Verified.

7.3.3 Missing events in `NodeOperatorRegistry` update methods

Description: The following functions in `NodeOperatorRegistry` update multiple storage locations but don't emit any events:

- `updateOperatorControllingAddress`
- `updateOperatorRewardAddress`
- `updateOperatorName`

Consider emitting events in these functions to reflect the updates made to storage.

Swell: Fixed in commit 5849640.

Cyfrin: Verified.

7.3.4 Refactor identical code in `NodeOperatorRegistry::getNextValidatorDetails`

Description: The bodies of these two `else if` branches are identical:

```
} else if (foundOperatorId == 0) {
  // If no operator has been found yet set the smallest operator active keys to the current operator
  smallestOperatorActiveKeys = operatorActiveKeys;

  foundOperatorId = operatorId;

  // If the current operator has less keys than the smallest operator active keys, then we want to use
  ↪  this operator
} else if (smallestOperatorActiveKeys > operatorActiveKeys) {
  smallestOperatorActiveKeys = operatorActiveKeys;

  foundOperatorId = operatorId;
}
```

Hence the code can be simplified to:

```
// If no operator has been found yet set the smallest operator active keys to the current operator
// If the current operator has less keys than the smallest operator active keys, then we want to use
↪  this operator
} else if (foundOperatorId == 0 ||
           smallestOperatorActiveKeys > operatorActiveKeys) {
  smallestOperatorActiveKeys = operatorActiveKeys;
  foundOperatorId = operatorId;
}
```

Swell: Fixed in commit d457d8d.

Cyfrin: Verified.

7.4 Gas Optimization

7.4.1 Cache storage variables in memory when read multiple times without being changed

Description: As reading from storage is considerably more expensive than reading from memory, cache storage variables in memory when read multiple times without being changed:

File: `NodeOperatorRegistry.sol`

```
// @audit cache `numOperators` in memory from storage
// to prevent reading same value multiple times
113:    uint128[] memory operatorAssignedDetails = new uint128[](numOperators + 1);
125:     for (uint128 operatorId = 1; operatorId <= numOperators; operatorId++) {

// @audit save incremented value in memory
// to prevent reading same value multiple times, eg:
// uint128 newNumOperators = ++numOperators;
305:    numOperators += 1;
// then use `newNumOperators` in L314,315
314:    getOperatorIdForAddress[_operatorAddress] = numOperators;
315:    getOperatorForOperatorId[numOperators] = operator;
// @audit `Operator` struct can also be initialized this way:
// getOperatorForOperatorId[numOperators] = Operator(true, _rewardAddress, _operatorAddress, _name, 0);

// @audit cache `getOperatorForOperatorId[operatorId].activeValidators`
660:    if (getOperatorForOperatorId[operatorId].activeValidators == 0) {
666:      getOperatorForOperatorId[operatorId].activeValidators - 1
```

File: `RepricingOracle.sol`

```
// @audit cache rate when checked after repricing and use
// cached version when processing withdrawals since the rate
// only changes during repricing which has already occurred
125:    if (swETHToETHRate > AccessControlManager.swETH().swETHToETHRate()) {
132:        AccessControlManager.swETH().swETHToETHRate() // The rate to use for processing withdrawals

// @audit cache `upgradeableRepriceSnapshot.meta.blockNumber` in memory from storage
// to prevent reading same value multiple times
290:    bool useOldSnapshot = upgradeableRepriceSnapshot.meta.blockNumber == 0;
294:      : upgradeableRepriceSnapshot.meta.blockNumber;

// @audit cache `maximumRepriceBlockAtSnapshotStaleness` in memory from storage
// to prevent reading same value multiple times
317:    if (snapshotStalenessInBlocks > maximumRepriceBlockAtSnapshotStaleness) {
320:        maximumRepriceBlockAtSnapshotStaleness
```

File: `swETH.sol`

```
// @audit cache `lastRepriceUNIX` in memory from storage
// to prevent reading same value multiple times
222:    uint256 timeSinceLastReprice = block.timestamp - lastRepriceUNIX;
249:    if (lastRepriceUNIX != 0) {

// @audit cache `minimumRepriceTime` in memory from storage
// to prevent reading same value multiple times
224:    if (timeSinceLastReprice < minimumRepriceTime) {
226:        minimumRepriceTime - timeSinceLastReprice

// @audit cache `nodeOperatorRewardPercentage` in memory from storage
// to prevent reading same value multiple times
233:      nodeOperatorRewardPercentage;
281:      UD60x18 nodeOperatorRewardPortion = wrap(nodeOperatorRewardPercentage)

// @audit cache `swETHToETHRateFixed` in memory from storage
// to prevent reading same value multiple times
253:        swETHToETHRateFixed
256:      uint256 maximumRepriceDiff = wrap(swETHToETHRateFixed)

// @audit no need to re-read storage values, use the in-memory variables
// that storage locations were just updated from to eliminate redundant but
// expensive storage reads
337:    lastRepriceETHReserves = totalReserves;
338:    lastRepriceUNIX = block.timestamp;
339:    swETHToETHRateFixed = updatedSwETHToETHRateFixed;

341:   emit Reprice(
342:      lastRepriceETHReserves, // @audit use `totalReserves` instead
343:       swETHToETHRateFixed,   // @audit use `updatedSwETHToETHRateFixed` instead
344:       nodeOperatorRewards,
345:       swellTreasuryRewards,
346:       totalETHDeposited

// @audit the first check will fail most of the time during regular usage so
// `swETHToETHRateFixed` will be read twice from storage with the same value
374:    if (swETHToETHRateFixed == 0) {
375:    return wrap(swETHToETHRateFixed);
```

File: `swEXIT.sol`

```
// @audit consider caching `withdrawRequestMinimum` and `withdrawRequestMaximum`
// in memory to avoid an extra storage read in the revert case
193:    if (amount < withdrawRequestMinimum) {
194:       revert WithdrawRequestTooSmall(amount, withdrawRequestMinimum);
195:     }

197:     if (amount > withdrawRequestMaximum) {
198:      revert WithdrawRequestTooLarge(amount, withdrawRequestMaximum);
199:     }
```

Swell: Fixed in commits 23be897, 3f85df3.

Cyfrin: Verified.

7.4.2 Cache array length outside of loops and consider unchecked loop incrementing

Description: Cache array length outside of loops and consider using `unchecked {++i;}` if not compiling with `solc --ir-optimized --optimize`:

File: `DepositManager.sol`

```
// @audit cache `validatorDetails.length`
116:    for (uint256 i; i < validatorDetails.length; i++) {
```

File: `NodeOperatorRegistry.sol`

```
// @audit cache `numOperators`
133:    uint128[] memory operatorAssignedDetails = new uint128[](numOperators + 1);
125:      for (uint128 operatorId = 1; operatorId <= numOperators; operatorId++) {

// @audit cache `_pubKeys.length`
189:    validatorDetails = new ValidatorDetails[](_pubKeys.length);
191:    for (uint256 i; i < _pubKeys.length; i++) {
227:    numPendingValidators -= _pubKeys.length;

// @audit cache `_validatorDetails.length`
243:    if (_validatorDetails.length == 0) {
257:        _validatorDetails.length >
263:    for (uint128 i; i < _validatorDetails.length; i++) {
282:    numPendingValidators += _validatorDetails.length;

// @audit cache `_pubKeys.length`
396:    for (uint128 i; i < _pubKeys.length; i++) {
412:     numPendingValidators -= _pubKeys.length;

// @audit cache `_pubKeys.length`
425:     for (uint256 i; i < _pubKeys.length; i++) {

// @audit cache `operatorIdToValidatorDetails[operatorId].length()`
628:     if (operatorIdToValidatorDetails[operatorId].length() == 0) {
634:      operatorIdToValidatorDetails[operatorId].length() - 1
```

File: `swEXIT.sol`

```
// @audit cache `requestsToProcess + 1`
143:    for (uint256 i = 1; i < requestsToProcess + 1; ) {
```

File: `Whitelist.sol`

```
// @audit cache `_addresses.length`
 84:    for (uint256 i; i < _addresses.length; ) {
102:    for (uint256 i; i < _addresses.length; ) {
```

Swell: Fixed in commits 3c67e88, 3f85df3.

Cyfrin: Verified.

7.4.3 `NodeOperatorRegistry::_parsePubKeyToString`: Use shift operations rather than division/multiplication when dividend/factor is a power of 2

Description: While `DIV` and `MUL` opcodes cost 5 gas unit each, shift operations cost 3 gas units. Therefore, `NodeOperatorRegistry::_parsePubKeyToString` can take advantage of them to save gas:

```
+   uint256 private SYMBOL_LENGTH = 16 // Because _SYMBOLS.length = 16
    function _parsePubKeyToString(
        bytes memory pubKey
    ) internal pure returns (string memory) {
        // Create the bytes that will hold the converted string
-       bytes memory buffer = new bytes(pubKey.length * 2);
+       // make sure that pubKey.length * 2 <= 2^256
+       bytes memory buffer = new bytes(pubKey.length << 1);

        bytes16 symbols = _SYMBOLS;
+       uint256 symbolLength = symbols.length;
+       uint256 index;
        for (uint256 i; i < pubKey.length; i++) {
-           buffer[i * 2] = symbols[uint8(pubKey[i]) / symbols.length];
-           buffer[i * 2 + 1] = symbols[uint8(pubKey[i]) % symbols.length];
+           index = i << 1; // i * 2
+           buffer[index] = symbols[uint8(pubKey[i]) >> 4]; // SYMBOL_LENGTH = 2^4
+           buffer[index + 1] = symbols[uint8(pubKey[i]) % SYMBOL_LENGTH];
        }

        return string(abi.encodePacked("0x", buffer));
    }
```

A more optimized version of this function looks like:

```
bytes16 private constant _SYMBOLS = "0123456789abcdef";
uint256 private constant SYMBOL_LENGTH = 16; // Because _SYMBOLS.length = 16

function _parsePubKeyToString(bytes memory pubKey) internal pure returns (string memory) {
    // Create the bytes that will hold the converted string
    // make sure that pubKey.length * 2 <= 2^256
    uint256 pubKeyLength  = pubKey.length;
    bytes memory buffer   = new bytes(pubKeyLength << 1);

    uint256 index;
    for (uint256 i; i < pubKeyLength;) {
        index             = i << 1; // i * 2
        buffer[index]     = _SYMBOLS[uint8(pubKey[i]) >> 4]; // SYMBOL_LENGTH = 2^4
        buffer[index + 1] = _SYMBOLS[uint8(pubKey[i]) % SYMBOL_LENGTH];

        unchecked {++i;}
    }

    return string(abi.encodePacked("0x", buffer));
}
```

The following stand-alone test using Foundry & Halmos verifies that the optimized version returns the same output as the original:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.23;

import "forge-std/Test.sol";

// run from base project directory with:
// halmos --function test --match-contract ParseTest
contract ParseTest is Test {

    bytes16 private constant _SYMBOLS = "0123456789abcdef";
    uint256 private constant SYMBOL_LENGTH = 16; // Because _SYMBOLS.length = 16

    function _parseOriginal(bytes memory pubKey) internal pure returns (string memory) {
        // Create the bytes that will hold the converted string
        bytes memory buffer = new bytes(pubKey.length * 2);

        bytes16 symbols = _SYMBOLS;

        // This conversion relies on taking the uint8 value of each byte, the first character in the
        ↪  byte is the uint8 value divided by 16 and the second character is modulo of the 16 division
        for (uint256 i; i < pubKey.length; i++) {
            buffer[i * 2] = symbols[uint8(pubKey[i]) / symbols.length];
            buffer[i * 2 + 1] = symbols[uint8(pubKey[i]) % symbols.length];
        }

        return string(abi.encodePacked("0x", buffer));
    }

    function _parseOptimized(bytes memory pubKey) internal pure returns (string memory) {
        // Create the bytes that will hold the converted string
        // make sure that pubKey.length * 2 <= 2^256
        uint256 pubKeyLength  = pubKey.length;
        bytes memory buffer   = new bytes(pubKeyLength << 1);

        uint256 index;
        for (uint256 i; i < pubKeyLength;) {
            index             = i << 1; // i * 2
            buffer[index]     = _SYMBOLS[uint8(pubKey[i]) >> 4]; // SYMBOL_LENGTH = 2^4
            buffer[index + 1] = _SYMBOLS[uint8(pubKey[i]) % SYMBOL_LENGTH];

            unchecked {++i;}
        }

        return string(abi.encodePacked("0x", buffer));
    }

    function test_HalmosParse(bytes memory pubKey) public {
        string memory resultOriginal  = _parseOriginal(pubKey);
        string memory resultOptimized = _parseOptimized(pubKey);

        assertEq(resultOriginal, resultOptimized);

    }
}
```

Swell: Fixed in commits 7db1874, 3f85df3.

Cyfrin: Verified.

7.4.4 Use `totalReserves - rewardsInETH.unwrap()` rather than `_preRewardETHReserves - rewardsInETH.unwrap() + _newETHRewards` in `swETH::reprice`

Description: Both result in the same output but the first expression saves a `SUB` opcode. In addition the suggested modification results in simpler code which better reflects the intention of the invariant.

```
    // swETH::reprice
    uint256 totalReserves = _preRewardETHReserves + _newETHRewards;

    uint256 rewardPercentageTotal = swellTreasuryRewardPercentage +
      nodeOperatorRewardPercentage;

    UD60x18 rewardsInETH = wrap(_newETHRewards).mul(
      wrap(rewardPercentageTotal)
    );

    UD60x18 rewardsInSwETH = wrap(_swETHTotalSupply).mul(rewardsInETH).div(
-       wrap(_preRewardETHReserves - rewardsInETH.unwrap() + _newETHRewards)
+       wrap(totalReserves - rewardsInETH.unwrap())
    );
```

Swell: Fixed in commit 7db1874.

Cyfrin: Verified.

7.4.5 Remove redundant pause checks

Description: 1) Remove redundant `botMethodsPaused` check in `swETH::reprice` as:

- this function is only called by `RepricingOracle::handleReprice`
- `RepricingOracle::handleReprice` can only be called by `submitSnapshot` and `submitSnapshotV2` which both already contain the `botMethodsPaused` check.

2) Remove redundant `withdrawalsPaused` check in `swEXIT::processWithdrawals` as this function is only supposed to be callable by `RepricingOracle` which already contains the check.

Swell: Fixed in commits 1fca965, 3f85df3.

Cyfrin: Verified.

7.4.6 Refactor `RepricingOracle::handleReprice`, `_assertRepricingSnapshotValidity` and `_repricingPeriodDeltas`

Description: In `RepricingOracle::_assertRepricingSnapshotValidity` and `_repricingPeriodDeltas` there is a lot of logic around whether to use the old snapshot or not, based around if `upgradeableRepriceSnapshot.meta.blockNumber == 0`.

If the idea is that the first time repricing occurs after the upgrade the execution path is `useOldSnapshot = true` but after that every time it will be `useOldSnapshot = false`, then it may make more sense to create functions just for that first execution which will only run once, then have functions for all the normal cases which come afterwards. This would avoid the extra gas costs and also simplify the code for all the future normal cases after the first-time-call special case.

Gas costs can also be reduced by having `handleReprice` load the snapshot struct, cache `upgradeableRepriceSnapshot.meta.blockNumber`, calculate `useOldSnapshot` once then pass these in as inputs to `_assertRepricingSnapshotValidity` and `_repricingPeriodDeltas` eg:

```
function handleReprice(
  UpgradeableRepriceSnapshot calldata _snapshot
) internal {
  // only call getSnapshotStruct() once
  UpgradeableRepriceSnapshot
    storage upgradeableRepriceSnapshot = getSnapshotStruct();

  // only calculate these once and pass them as required
  uint256 ursMetaBlockNumber = upgradeableRepriceSnapshot.meta.blockNumber;
  bool useOldSnapshot = ursMetaBlockNumber == 0;

  // validation
  _assertRepricingSnapshotValidity(_snapshot, ursMetaBlockNumber, useOldSnapshot);

  _repricingPeriodDeltas(
        reserveAssets,
        _snapshot.state,
        _snapshot.withdrawState,
        upgradeableRepriceSnapshot,
        useOldSnapshot
      );

  // delete the call to getSnapshotStruct() near the end of handleReprice()
```

Swell: Acknowledged. Will be addressed in a future upgrade when the old snapshot is no longer relevant. Swell will continue to pay the excess gas costs in the meantime.

7.4.7 Use constant for unchanging deposit amount

Description: In `DepositManager::setupValidators` there is no use in paying gas to declare then later read this variable which never changes:

```
uint256 depositAmount = 32 ether;
```

Rather simply define a constant:

```
uint256 private constant DEPOSIT_AMOUNT = 32 ether;
```

And use that constant instead.

Swell: Acknowledged.