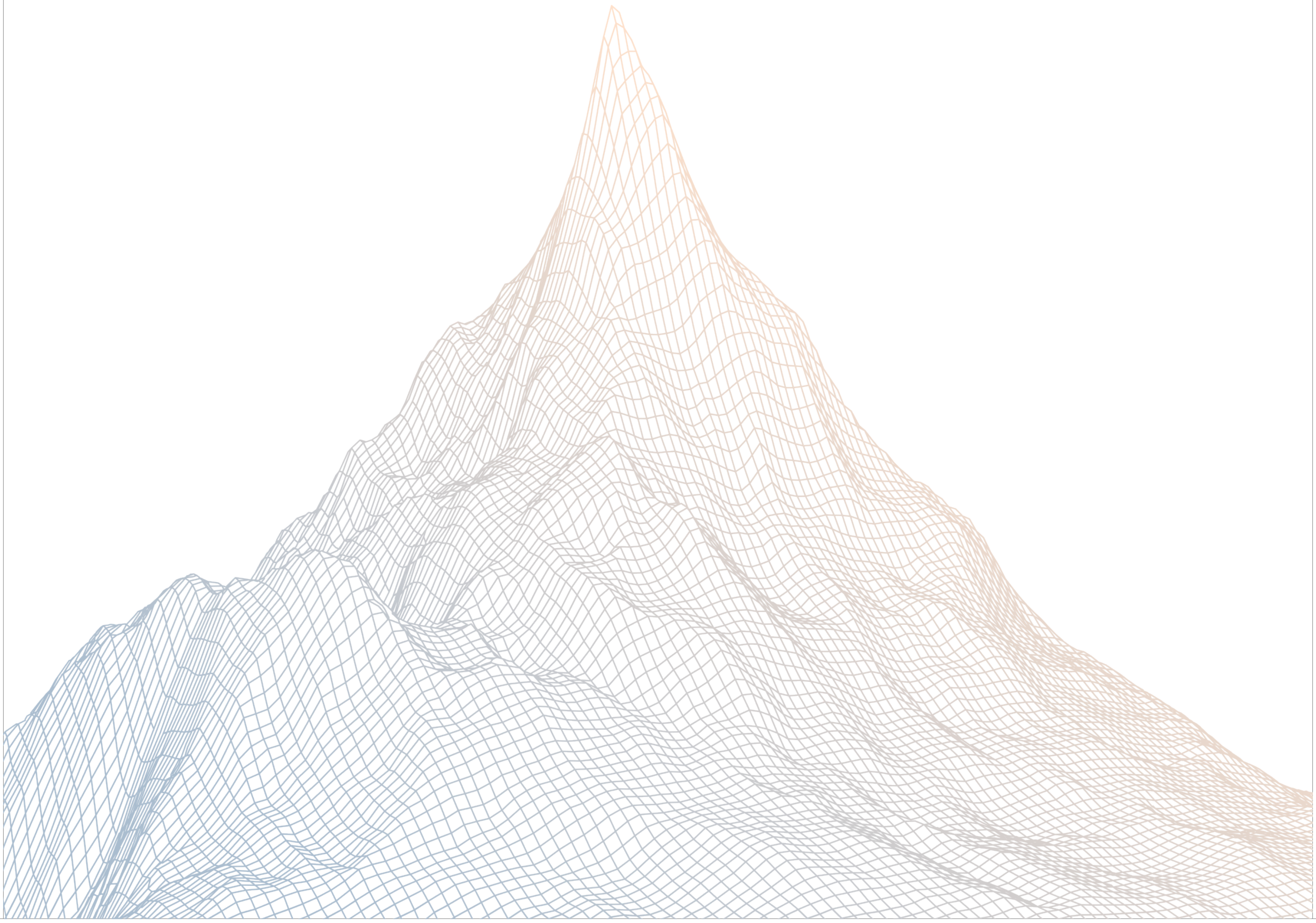
Zenith

Swell

Smart Contract Security Assessment

VERSION 1.1



AUDIT DATES: July 2nd to July 7th, 2025

AUDITED BY: rscodes
windhustler

Contents

1 Introduction 2

1.1 About Zenith 3

1.2 Disclaimer 3

1.3 Risk Classification 3

2 Executive Summary 3

2.1 About Swell 4

2.2 Scope 4

2.3 Audit Timeline 5

2.4 Issues Found 5

2.5 Security Note 5

3 Findings Summary 5

4 Findings 6

4.1 Critical Risk 7

4.2 High Risk 9

4.3 Proof of Concept 11

4.4 Medium Risk 13

4.5 Low Risk 24

4.6 Informational 25

1 Introduction

1.1 About Zenith

Zenith assembles auditors with proven track records: finding critical vulnerabilities in public audit competitions.

Our audits are carried out by a curated team of the industry's top-performing security researchers, selected for your specific codebase, security needs, and budget.

Learn more about us at https://zenith.security.

1.2 Disclaimer

This report reflects an analysis conducted within a defined scope and time frame, based on provided materials and documentation. It does not encompass all possible vulnerabilities and should not be considered exhaustive.

The review and accompanying report are presented on an "as-is" and "as-available" basis, without any express or implied warranties.

Furthermore, this report neither endorses any specific project or team nor assures the complete security of the project.

1.3 Risk Classification

SEVERITY LEVEL	IMPACT: HIGH	IMPACT: MEDIUM	IMPACT: LOW
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

2 Executive Summary

2.1 About Swell

Swell is a non-custodial staking protocol with a mission to deliver the world's best liquid staking and restaking experience, simplify access to DeFi, while securing the future of Ethereum and restaking services.

With Swell, users are able to earn passive income by staking or restaking ETH to earn both blockchain rewards and restaked AVS rewards, and in return be provided with a yield-bearing liquid token (LST or LRT) to hold or participate in the wider DeFi ecosystem to earn additional yield.

2.2 Scope

The engagement involved a review of the following targets:

Target	swhype
Repository	https://github.com/SwellNetwork/swhype/
Commit Hash	`1e09878b83261e808e3fba5a05ca10611d49a082`

Files	`HypeStakingContract.sol`
	`sswHYPE.sol`
	`swHYPE.sol`
	`xERC4626.sol`

2.3 Audit Timeline

July 2, 2025	Audit start
July 7, 2025	Audit end
July 22, 2025	Report published

2.4 Issues Found

SEVERITY	COUNT
Critical Risk	1
High Risk	2
Medium Risk	6
Low Risk	1
Informational	3
Total Issues	**13**

2.5 Security Note

Considering the number of issues identified, it is statistically likely that there are more complex bugs still present that could not be identified given the time-boxed nature of this engagement. It is recommended that a follow-up audit and development of a more complex stateful test suite be undertaken prior to continuing to deploy significant monetary capital to production.

3 Findings Summary

ID	Description	Status
C-1	Victim's asset tokens could be stolen on source chain after bridging due to incorrect ratio	Resolved
H-1	Reward tokens could be spread across wrong duration due to lastsync not updating	Resolved
H-2	Queued rewards immediately included in totalAssets calculation leads to bypassing linear reward distribution	Resolved
M-1	Redundant and incorrect message validation in lzReceive	Resolved
M-2	queueRewards not calling syncRewards means rewards could end up in wrong cycle	Resolved
M-3	Operator can mint unbacked swHYPE tokens	Acknowledged
M-4	Incorrect ERC4626 rounding direction leads to potential vault value loss	Resolved
M-5	ERC4626 inflation attack in sswHYPE	Resolved
M-6	Permit frontrunning enables denial of service attacks	Resolved
L-1	No way for admin to change rewardsCycleLength	Resolved
I-1	No slippage option when using HypeStakingContract for staking in sswHypeVault	Resolved
I-2	Redundant if statement check	Resolved
I-3	Missing swHYPE redemption mechanism leads to permanent user fund lockup	Acknowledged

4 Findings

4.1 Critical Risk

A total of 1 critical risk findings were identified.

[C-1] Victim's asset tokens could be stolen on source chain after bridging due to incorrect ratio

SEVERITY: Critical	IMPACT: High
STATUS: Resolved	LIKELIHOOD: High

Target

- sswHYPE.sol

Description:

In the vault `sswHYPE.sol`, when users bridge out of the source chain, the code burns the shares but `totalAssets()` remains the same.

This can be seen in `_debit`:

```
function _debit(address _from, uint256 _amountLD, uint256 _minAmountLD,
    uint32 _dstEid)
    internal
    virtual
    override(OFTUpgradeable)
    whenNotPaused
    returns (uint256 amountSentLD, uint256 amountReceivedLD)
{
    (amountSentLD, amountReceivedLD) = _debitView(_amountLD, _minAmountLD,
    _dstEid);
->  _burn(_from, amountSentLD);
    emit CrossChainMessageSent(_dstEid, abi.encode(amountSentLD, _from));
    return (amountSentLD, amountReceivedLD);
}
```

We can see that the shares bridged away are burned but `totalAssets()` remains the same, hence **wrongly inflating the ratio**.

Consider this sequence where the victim loses assets:

1. Alice and Bob are both stake `10e18` asset tokens each.
2. Alice chooses to bridge her shares away to another destination chain.
3. `sswHYPE` burns her shares (decreasing `totalSupply()`) but does not update `totalAssets()`.
4. Hence `totalAssets()` remains the same at `20e18`.
5. Now this inflates Bob's share ratio since `totalSupply()` got burned while `totalAssets()` stayed the same.
6. Bob takes the opportunity to withdraw and steals Alice's asset tokens.

Recommendations:

Keep track of the corresponding assets amount bridged away, then lock them. After that, override `totalAssets()` to subtract away the locked amounts.

```
function totalAssets() public view override returns (uint256) {
    return super.totalAssets() - assetLocked;
}
```

Swell: Resolved with @99c42e5fe64...

Zenith: Verified.

4.2 High Risk

A total of 2 high risk findings were identified.

[H-1] Reward tokens could be spread across wrong duration due to lastsync not updating

SEVERITY: High	IMPACT: High
STATUS: Resolved	LIKELIHOOD: Medium

Target

- xERC4626.sol

Description:

`syncRewards` is the function that is responsible for updating `lastSyncTime`. However, there is an edge case that if the `queued` is currently zero, then `lastSyncTime` does not update.

It is still important to update it even though `queued` is currently zero as queued rewards may be added halfway through the cycle, resulting in the rewards being wrongly spread across the first half of the cycle as well.

Consider this sequence.

1. The reward cycle is 1 week.
2. For the first 3 days, `queued == 0`, hence `syncRewards` simply doesn't update `lastSyncTime`.
3. At the 4th day, `10e18` of rewards are added.
4. Even if the admin purposely called `syncRewards()` right before the addition of the rewards, `lastSyncTime` won't be updated as `queued == 0`.
5. On the 5th day, a staker tries to withdraw. The reward they will get will be calculcated as `10e18 * 5/7` even though it should only be `10e18 * 2/4`. (as they have only been accumulating on day3 and 4).

Recommendations:

Add an `else` after the else if in `syncRewards` so that `lastSyncTime` can be updated even if `queued` is zero.

```
function syncRewards() public {
  if (timestamp >= cycleEnd) {
    ...
  } else if (queued > 0 && timestamp > lastSyncTime) {
    ...
  } else {
    _updatePackedRewardsState(timestamp, lastRewardAmount());
  }
}
```

Swell: Resolved with PR-14

Zenith: Verified.

[H-2] Queued rewards immediately included in totalAssets calculation leads to bypassing linear reward distribution

SEVERITY: High	IMPACT: High
STATUS: Resolved	LIKELIHOOD: High

Target

- xERC4626.sol

Description:

The xERC4626 contract implements a linear reward distribution mechanism intended to prevent MEV attacks by distributing rewards over time. However, this mechanism is fundamentally broken because queued rewards are immediately included in the `totalAssets()` calculation, allowing users to bypass the linear distribution and claim rewards immediately.

When `xERC4626::queueRewards` is called, the reward tokens are transferred directly to the vault contract. Since `totalAssets()` returns `_asset.balanceOf(address(this))`, these queued rewards immediately become part of the total assets calculation, making them available for withdrawal/redemption at the new exchange rate.

The `syncRewards()` function only updates storage variables (`_packedRewardsData` and `_packedRewardsState`) but does not affect the actual reward distribution mechanism, as the underlying asset balance remains unchanged.

4.3 Proof of Concept

1. Initial state: `totalSupply() = 100`, `totalAssets() = 100`
2. Owner calls `xERC4626::queueRewards(10)` - transfers 10 reward tokens to the vault
3. Now `totalAssets() = 110` (because `totalAssets()` returns `_asset.balanceOf(address(this))`)
4. User immediately calls `redeem(10, user, user)` and receives `10 * 110 / 100 = 11` tokens
5. User has bypassed the linear distribution and claimed rewards immediately

6. Calling `syncRewards()` only updates storage variables but doesn't affect the actual reward distribution

Recommendations:

`totalAssets` should only include the rewards that were distributed according to the linear vesting schedule.

Swell: Resolved with PR-13

Zenith: Verified. Issue has been fixed by keeping track of unvested amounts and subtracting them.

4.4 Medium Risk

A total of 6 medium risk findings were identified.

[M-1] Redundant and incorrect message validation in lzReceive

`SEVERITY:` Medium	`IMPACT:` Medium
`STATUS:` Resolved	`LIKELIHOOD:` Low

Target

- sswHYPE.sol#L312
- swHYPE.sol#L137

Description:

The `swHYPE::_lzReceive` function implements multiple layers of redundant validation that are already handled by LayerZero's core infrastructure.

The function performs several redundant checks:

1. **Message length validation (`_message.length < 4`)** - Already validated in `OFTCore.sol` since it extracts `toAddress` and `amountSD` from the message, which requires >4 bytes.
2. **Source endpoint ID validation (`_origin.srcEid == 0`)** - Already enforced by LayerZero endpoint and checked in `OAppReceiverUpgradeable.lzReceive()`.
3. **Executor address validation** - Contains an impossible condition since executor cannot be `address(0)` and message length is always >0 due to prior checks.
4. **Message format validation** - Uses incorrect decoding logic that will always fail for valid LayerZero messages.

The most critical issue is in point 4: the `validateMessageFormat` function attempts to decode LayerZero's packed message format incorrectly, which would always revert if reached.

LayerZero's `OFTMsgCodec.encode` creates messages using `abi.encodePacked(_sendTo, _amountShared)` producing:

- 32 bytes: `_sendTo` (bytes32)

- 8 bytes: `_amountShared` (uint64)
- **Total: 40 bytes** (tightly packed)

However, `swHYPE::validateMessageFormat` attempts to decode using `abi.decode(_message, (uint256, address))`, expecting:

- 32 bytes: amount (uint256)
- 32 bytes: recipient (address)
- **Total: 64 bytes** (ABI-encoded with padding)

All the conditions described above apply to the `sswHYPE` contract as well.

Recommendations:

Remove all the redundant checks, and if you need to retrieve data from the message always use the `OFTMsgCodec` library from LayerZero for proper decoding.

Swell: Resolved with PR-13

Zenith: Verified.

[M-2] `queueRewards` not calling `syncRewards` means rewards could end up in wrong cycle

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Low

Target

- xERC4626.sol

Description:

```
// in queueRewards
_asset.safeTransferFrom(_msgSender(), address(this), amount);

uint192 newQueued = currentQueued + uint192(amount);
_updatePackedRewardsData(rewardsCycleLength(), rewardsCycleEnd(),
    newQueued);

unchecked {
    totalRewards += amount;
}
```

We can see that in `queueRewards` of `xERC4626.sol` there is no call to `syncRewards` and also no check if the current cycle is an **outdated** one which already ended. Rather the tokens are transferred in and the reward states are immediately updated.

This means that if it is currently supposed to be a transition between reward cycles then the reward amount might be added to the wrong cycle.

For example, if the current cycle has ended (which means `rewardsCycleEnd() < block.timestamp`), `queueRewards` will add the reward tokens as per normal to `newQueued`, instead of starting a new cycle (which should rightfully be the case since the current one ended even before the new rewards are added).

- If we take a look at `syncRewards`, rewards from cycles which ended are just released all at once. Hence, the new rewards added won't be distributed as part of a new cycle but rather it will just be batched together with a cycle that ended even before the rewards were added.

Recommendations:

If the reward cycle has ended, start a new cycle and include the new rewards under the new cycle.

Swell: Resolved with PR-16

Zenith: Verified. Issue has been fixed by adding a `syncRewards()` call into `queueRewards`.

[M-3] Operator can mint unbacked swHYPE tokens

`SEVERITY:` Medium	`IMPACT:` Medium
`STATUS:` Acknowledged	`LIKELIHOOD:` Low

Target

- HypeStakingContract.sol#L446

Description:

The `HypeStakingContract::mintRewardsSwHype` function allows the operator to mint swHYPE tokens without providing any corresponding HYPE tokens as backing. This breaks the fundamental 1:1 backing ratio between HYPE and swHYPE tokens, systematically devaluing all existing swHYPE holders.

The function is designed to represent "compounded rewards on Hyperliquid" but fails to ensure that the minted swHYPE tokens are actually backed by equivalent HYPE reserves.

Recommendations:

Modify the `HypeStakingContract::mintRewardsSwHype` function to require the operator to deposit equivalent HYPE tokens before minting swHYPE.

Swell: Partially resolved with PR-17.

I've updated the PR and would like to provide more context on this finding, as we believe it's a clarification of the design pattern rather than an issue. We are following a similar design to Frax with frxEth and sfrxEth (in our case swHype and sswHype respectively). To maintain the strict 1:1 peg of swHype to the underlying Hype, the protocol must mint new swHype equivalent to the staking rewards earned on HyperCore. This new supply is not unbacked; it's backed by the accrued rewards. The value from this yield is then immediately transferred to the sswHype vault, causing the value of sswHype to increase. To note any staking rewards are restaked back on HyperCore and they are not withdrawn.

The update I did in the PR combines the mint & deposit part into one function providing atomicity.

Additionally, replaced deposit with a call to queueRewards.

Zenith: Acknowledged. The distributeRewardsToVault() function mints swHYPE tokens representing staking rewards earned on HyperCore and deposits them into the sswHYPE vault, but it does not transfer the corresponding HYPE rewards from HyperCore to the staking contract to back the newly minted tokens. This is expected to occur separately.

[M-4] Incorrect ERC4626 rounding direction leads to potential vault value loss

`SEVERITY:` Medium	`IMPACT:` Medium
`STATUS:` Resolved	`LIKELIHOOD:` Medium

Target

- sswHYPE.sol
- xERC4626.sol

Description:

The `xERC4626` contract implements ERC4626 preview functions with incorrect rounding directions, violating the ERC4626 specification. According to EIP-4626, preview functions must round in favor of the vault to prevent economic attacks, but the current implementation rounds down for all operations.

The affected functions that should round up but currently round down are:

- `xERC4626::previewMint`
- `xERC4626::previewWithdraw`

Additionally, the corresponding implementation functions in the inheriting `sswHYPE` contract must also be updated to maintain consistency between preview and actual execution:

- `mint` function should use rounding up when converting shares to assets
- `withdraw` function should use rounding up when converting assets to shares

Recommendations:

Update the `previewMint`, `previewWithdraw`, `mint`, and `withdraw` functions to round in favor of the vault.

Swell: Resolved with PR-19

Zenith: Verified

[M-5] ERC4626 inflation attack in sswHYPE

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Low

Target

- sswHYPE.sol#L170](https://github.com/SwellNetwork/swhype/blob/1e09878b83261e808e3fba

Description:

The `sswHYPE::deposit` function is vulnerable to the ERC4626 inflation attack, a well-known vulnerability in vault contracts that allows an attacker to manipulate the share price and steal user deposits through precision loss. Even with the existing `ZeroValue()` protection that reverts when `shares == 0`, the attack can still be executed to cause significant losses.

The attack exploits the conversion between assets and shares in the `convertToShares` function, which calculates `assets.mulDiv(supply, totalAssets())`. An attacker can manipulate this calculation by inflating the `totalAssets()` value while keeping `totalSupply()` minimal, causing victim deposits to round down to only 1 share instead of the proportional amount they should receive.

Proof of Concept

Following the "Rounding to one share" attack pattern:

1. Attacker back-runs the vault deployment when `totalSupply() == 0`.
2. Attacker calls `sswHYPE::deposit(1)` to mint 1 share for themselves. Now `totalAssets() == 1` and `totalSupply() == 1`.
3. Attacker monitors the mempool for a victim's large deposit transaction (e.g., 20,000 swHYPE tokens).
4. Attacker front-runs the victim by directly transferring a large amount of swHYPE (e.g., 10,000e18) to the vault contract, bypassing the deposit function. Now `totalAssets() == 10,000e18 + 1` while `totalSupply()` remains 1.
5. Victim's transaction executes: `convertToShares(20,000e18)` calculates `20,000e18 * 1 / (10,000e18 + 1) ≈ 1 share` due to rounding down.

6. The victim receives only 1 share (same as the attacker) but deposits 20,000e18 swHYPE tokens.
7. Attacker redeems their 1 share and receives approximately `(10,000e18 + 1 + 20,000e18) / 2 = 15,000e18` swHYPE tokens.
8. Attacker's profit is approximately 5,000e18 swHYPE tokens (25% of victim's deposit).

Recommendations:

Users should be advised to always use the `depositWithSlippage` function. Otherwise, protection for the slippage attack like minting dead shares or implementing the virtual offset should be used.

Swell: Resolved with @2eb4e20dcd...

Zenith: Verified.

[M-6] Permit frontrunning enables denial of service attacks

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Low

Target

- HypeStakingContract.sol#L308
- sswHYPE.sol#L276

Description:

The `HypeStakingContract::stakeHypeWithPermit` function is vulnerable to permit frontrunning attacks that can cause a denial of service. The function directly calls `IERC20Permit::permit` without handling the case where the permit has already been used, making it susceptible to griefing attacks.

When a user submits a transaction calling `stakeHypeWithPermit`, an attacker can observe the transaction in the mempool and extract the permit parameters (amount, deadline, v, r, s). The attacker then frontruns the original transaction by calling `permit` directly on the token contract with the same parameters. When the user's original transaction is executed, the `permit` call reverts because the nonce has already been used, causing the entire staking operation to fail.

More on this bug class can be read in this blog post.

The same attack can be executed with `sswHYPE::depositWithPermit`.

Recommendations:

Wrap the permit call in a try-catch block to handle the case where the permit has already been used:

```
IERC20Permit(address(hypeToken)).permit(msg.sender, address(this), amount,
    deadline, v, r, s);
try IERC20Permit(address(hypeToken)).permit(msg.sender, address(this),
    amount, deadline, v, r, s) {
    // Permit successful
} catch {
    // Permit failed, check if allowance is sufficient
```

```
        if (hypeToken.allowance(msg.sender, address(this)) < amount) {
            revert("Insufficient allowance");
        }
    }
```

Swell: Resolved with PR-21

Zenith: Verified.

4.5 Low Risk

A total of 1 low risk findings were identified.

[L-1] No way for admin to change `rewardsCycleLength`

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- xERC4626.sol

Description:

In `xERC4626.sol`, the contract only has view-only functions regarding `rewardsCycleLength`:

```
function rewardsCycleLength() public view returns (uint32) {
    return uint32(_packedRewardsData);
}
```

Just like how rewards queued can be incremented by the admin depositing more tokens, it would be more flexible if the contract could have a function which allows the admin to change and adjust the reward cycle lengths as well.

Recommendations:

Consider letting admin adjust reward cycle lengths.

Swell: Resolved with PR-22

Zenith: Verified.

4.6 Informational

A total of 3 informational findings were identified.

[I-1] No slippage option when using `HypeStakingContract` for staking in sswHypeVault

`SEVERITY`: Informational	`IMPACT`: Informational
`STATUS`: Resolved	`LIKELIHOOD`: Low

Target

- HypeStakingContract.sol

Description:

When staking in the sswHypeVault we can see that there are options like `depositWithSlippage` which allows the user to specify their allowed slippage tolerance.

However, if the user would to do it using the `HypeStakingContract.sol`, there are no such options as functions like `stakeSwHype(uint256 amount, address recipient)` and `submitAndDeposit(uint256 hypeAmount, address recipient)` **only calls** `sswHypeVault`'s `deposit`.

Recommendations:

Introduce an option for users to call `sswHypeVault`'s `depositWithSlippage` using `HypeStakingContract.sol`

Swell: Resolved with PR-13

Zenith: Verified.

[I-2] Redundant if statement check

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- xERC4626.sol#L136-L138

Description:

In `xERC4626.sol`:

```
function syncRewards() public {
    ....

    if (timestamp >= cycleEnd) {
        ...
    } else if (queued > 0 && timestamp > lastSyncTime) {
        unchecked {
          uint256 timeElapsed = timestamp - lastSyncTime;
          uint256 totalTime = cycleEnd - lastSyncTime;

          ....

          uint256 rewardAmount = (queued * timeElapsed) / totalTime;

          // Ensure rewardAmount doesn't exceed queued rewards
->        if (rewardAmount > queued) {
->            rewardAmount = queued;
->        }
          ....
```

The `rewardAmount > queued` check is redundant (and hence wastes gas) because since the code goes into the `else` part, `timestamp` is always < `cycleEnd`.

That means `totalTime ≥ timeElapsed`, hence `uint256 rewardAmount = (queued * timeElapsed) / totalTime` will mean `rewardAmount` will **always** be <= `queued`.

Recommendations:

Consider removing redundant if statement to save gas costs.

Swell: Resolved with PR-15

Zenith: Verified.

[I-3] Missing swHYPE redemption mechanism leads to permanent user fund lockup

`SEVERITY:` Informational	`IMPACT:` Informational
`STATUS:` Acknowledged	`LIKELIHOOD:` Low

Target

- swHYPE.sol

Description:

The `HypeStakingContract` allows users to stake HYPE tokens and receive swHYPE tokens at a 1:1 ratio through functions like `HypeStakingContract::stakeHype`, but provides no mechanism for users to redeem their swHYPE tokens back to HYPE. This creates a one-way flow where users cannot exit their positions, effectively locking their funds permanently in the protocol.

The system implements the following flows:

- HYPE → swHYPE via `HypeStakingContract::stakeHype`
- swHYPE → sswHYPE via `sswHYPE::deposit` / `sswHYPE::mint`
- sswHYPE → swHYPE via `sswHYPE::withdraw/sswHYPE::redeem`
- swHYPE → HYPE (missing)

Recommendations:

Implement a redemption function that burns `swHYPE` and returns HYPE at a 1:1 ratio.

Swell: Acknowledged.

Zenith: The issue was acknowledged since the `HypeStakingContract` is upgradeable, and the team wants to roll out redemptions in phases by upgrading the contracts.