hexens × yelay

SEPT.24

SECURITY REVIEW REPORT FOR YELAY

CONTENTS

- Merging the YelayOwnable and YelayOwner contracts for gas optimization
- Using console in the production code
- Discrepancies in sYLAYBase docs
- ConvertLib in-line calculations with literals should be replaced by constants
- Push-based user migration is vulnerable to sybil attack

EXECUTIVE SUMMARY

OVERVIEW

This audit covered an update to the governance contracts of Yelay, formerly known as Spool. New smart contracts and updates to old ones were introduced to facilitate the migration from SPOOL to YLAY and voSPOOL to sYLAY.

Our security assessment was a full review of the updates made, spanning a total of 1.5 weeks.

During our audit, we have identified 3 high severity vulnerabilities. These could have potentially caused loss of assets or caused the migration to fail.

We have also identified several minor severity vulnerabilities and code optimizations.

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.

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.

SCOPE

The analyzed resources are located on:

https://github.com/solidant/yelay/tree/2a914e088cb29fb8ec8cc495804822967c3b3f90

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

https://github.com/solidant/yelay/tree/68288fce52e823f37f2ff53b18c838dae511e1ae

AUDITING DETAILS

STARTED
16.09.2024

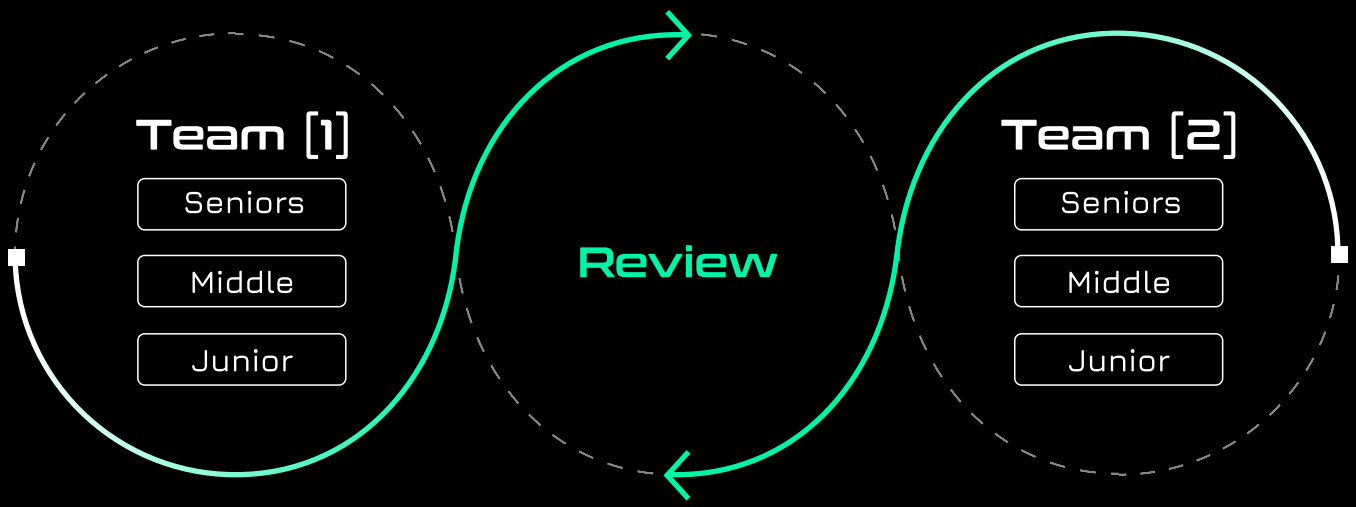
DELIVERED
04.10.2024

Review Led by

KASPER ZWIJSEN
Head of 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.

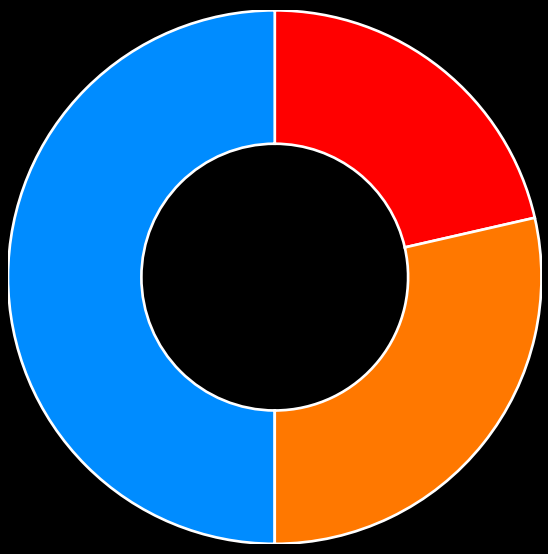
ISSUE SYMBOLIC CODES

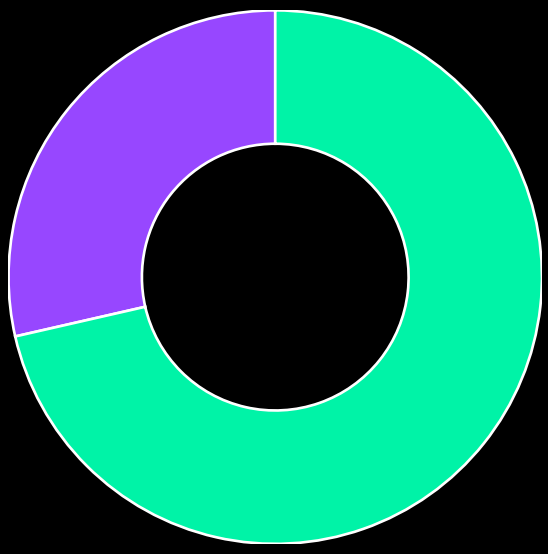
Every issue being identified and validated has its unique symbolic code assigned to the issue at the security research stage. Cause of the vulnerability reporting flow design, some of the rejected issues could be missing.

FINDINGS SUMMARY

Severity	Number of Findings
Critical	0
High	3
Medium	4
Low	0
Informational	7

Total: 14





WEAKNESSES

This section contains the list of discovered weaknesses.

YELAY8-10

MISSING UPDATE OF REWARD CONFIG FOR THE YELAYSTAKING CONTRACT DURING MIGRATEUSER FUNCTION

SEVERITY: High

PATH:

src/YelayStaking.sol#L78-L93

REMEDIATION:

migrateUser function of YelayStaking contract should trigger updateRewards modifier.

STATUS: Acknowledged, see commentary

DESCRIPTION:

In the YelayStakingBase contract, **updateRewards** is triggered whenever **totalStaked** is about to change. It triggers the **_updateReward** function to update the reward configuration for the latest period.

```
function _updateReward(IERC20 token, address account) private {
    RewardConfiguration storage config = rewardConfiguration[token];
    config.rewardPerTokenStored = rewardPerToken(token);
    config.lastUpdateTime = lastTimeRewardApplicable(token);
    if (account != address(0)) {
        config.rewards[account] = earned(token, account);
        config.userRewardPerTokenPaid[account] = config.rewardPerTokenStored;
    }
}
```

This is because **rewardPerToken()** is calculated based on the distribution among **totalStaked**, leading to different time periods having different **rewardPerTokenStored** configurations.

```
function rewardPerToken(IERC20 token) public view returns (uint224) {
    RewardConfiguration storage config = rewardConfiguration[token];

    if (totalStaked == 0) return config.rewardPerTokenStored;

    uint256 timeDelta = lastTimeRewardApplicable(token) -
config.lastUpdateTime;

    if (timeDelta == 0) return config.rewardPerTokenStored;

    return SafeCast.toUint224(config.rewardPerTokenStored + ((timeDelta *
config.rewardRate) / totalStaked));
}
```

However, during the **migrateUser** function in the YelayStaking contract, the reward configuration is not updated, resulting in **rewardPerTokenStored** not changing between user migrations. This causes the reward distribution to be unfair, leading to losses for some users, as later stakers can claim similar rewards to earlier ones.

Consider the following scenario:

1. At the beginning of day 1, Alice migrates a staking balance of 100 ether, making the total staked in YelayStaking 100 ether. The reward configuration is not updated.
2. At the beginning of day 2, Bob migrates a staking balance of 100 ether, making the total staked in YelayStaking 200 ether. The reward configuration is still not updated.
3. At the beginning of day 3, Bob unstakes his staking, which triggers the reward configuration update before execution. However, since **rewardPerTokenStored** was not updated earlier, Bob's rewards will be equal to Alice's rewards.

```
function migrateUser(address user) external onlyMigrator returns (uint256
yelayStaked, uint256 yelayRewards) {
    uint256 spoolStaked = spoolStaking.balances(user);
    yelayStaked = ConversionLib.convert(spoolStaked);

    YelayStakingMigrationStorage storage $ =
_getYelayStakingMigrationStorageLocation();
    unchecked {
        $.totalStakedSPOOLMigrated += spoolStaked;
    }

    _migrateUser(user, yelayStaked);

    uint256 userSpoolRewards = spoolStaking.earned(SPOOL, user);
    // SpoolStaking should be updated to SpoolStakingMigration before - to
facilitate migration
    uint256 userVoSpoolRewards =
SpoolStakingMigration(address(spoolStaking)).getUpdatedVoSpoolRewardAmount(u
ser);
    yelayRewards = ConversionLib.convert(userSpoolRewards +
userVoSpoolRewards);
}
```

Commentary from the client:

" - Rewards are to be added later. As there is no reward config (i.e., no **rewardRate**), **rewardPerToken** and **earned** will always return **0** for every user; hence, there is no benefit to add the modifier for the migration phase. It is the admin's responsibility to migrate all user stakes, which will be performed inside the same tranche window. So for practical purposes, the example doesn't apply. Adding it would unnecessarily increase the execution cost of the migration."

YELAY8-11

POTENTIAL LOSS OF USER FUNDS DURING SPOOL TO YLAY MIGRATION

SEVERITY: High

PATH:

src/YelayStaking.sol::_migrateUser()#L132-L139

REMEDIATION:

See description.

STATUS: Fixed

DESCRIPTION:

he **YelayStaking.sol** contract's migration process from **SPOOL** to **YLAY** tokens contains a flaw that could result in the loss of user funds. The **_migrateUser()** function, which is called during the migration process in the **migrateUser()**:

```
    function migrateUser(address user) external onlyMigrator returns
(uint256 yelayStaked, uint256 yelayRewards) {
        uint256 spoolStaked = spoolStaking.balances(user);
        yelayStaked = ConversionLib.convert(spoolStaked);

        YelayStakingMigrationStorage storage $ =
_getYelayStakingMigrationStorageLocation();
        unchecked {
            $.totalStakedSPOOLMigrated += spoolStaked;
        }

@>      _migrateUser(user, yelayStaked);
```

```
        uint256 userSpoolRewards = spoolStaking.earned(SPOOL, user);
        // SpoolStaking should be updated to SpoolStakingMigration before -
to facilitate migration
        uint256 userVoSpoolRewards =
SpoolStakingMigration(address(spoolStaking)).getUpdatedVoSpoolRewardAmount(u
ser);
        yelayRewards = ConversionLib.convert(userSpoolRewards +
userVoSpoolRewards);
    }
```

overwrites the user's existing YLAY balance instead of adding to it. This means that if a user has any existing YLAY tokens staked in the contract before the migration occurs, these tokens will be lost and replaced by the newly migrated amount.

```
    function _migrateUser(address account, uint256 amount) private {
        unchecked {
            totalStaked = totalStaked += amount;
        }
        balances[account] = amount;

        emit Staked(account, amount);
    }
```

Add the migrated amount to the existing balance rather than overwriting it:

```
    function _migrateUser(address account, uint256 amount) private {
        unchecked {
            totalStaked = totalStaked += amount;
+           balances[account] += amount;
        }
-       balances[account] = amount;

        emit Staked(account, amount);
    }
```

YELAY8-18

INCORRECT UPDATE OF LATEST GLOBAL TRANCHE INDEX DURING MIGRATION

SEVERITY: High

PATH:

src/sYLAY.sol:migrateGlobalBranches#L107

REMEDIATION:

In the function migrateGlobalTranches, line 107 should be corrected to:

$.lastGlobalIndexVoSPOOLMigrated = endIndex;

STATUS: Fixed

DESCRIPTION:

The function that migrates all global tranches from voSPOOL to sYLAY takes a parameter **endIndex** that is used inside of the for-loop to retrieve and set a global tranche.

The very first call will happen without problem, however after the for-loop, it increments the variable **lastGlobalIndexVoSPOOLMigrated** with **endIndex * 5** and this variable is used as the start index for the next call.

This incrementing is wrong, it is an index and so it should not use += as that would set the result to a higher value than **endIndex** if the start is greater than 0. Moreover, it should also not multiply **endIndex** with **TRANCHES_PER_WORD** (5) because it uses this index in the **indexedGlobalTranches** mapping which already holds 5 tranches per index.

This case will occur if the migration happens in batches, which is likely given the gas limitations of Ethereum mainnet. In such a case, the **lastGlobalIndexVoSPOOLMigrated** will grow too fast and so a lot of global branches will be skipped, leading to a failed migration.

```
    function migrateGlobalTranches(uint256 endIndex) external
migrationInProgress onlyMigrator {
        sYLAYMigrationStorage storage $ =
_getsYLAYMigrationStorageLocation();
        _whenInitialized($);
        if (_globalMigrationComplete()) return;

        for (uint256 i = $.lastGlobalIndexVoSPOOLMigrated; i < endIndex; i+
+) {
            (Tranche memory zero, Tranche memory one, Tranche memory two,
Tranche memory three, Tranche memory four) =
                voSPOOL.indexedGlobalTranches(i);
            (uint48 a, uint48 b, uint48 c, uint48 d, uint48 e) =
                ConversionLib.convertAmount(zero.amount, one.amount,
two.amount, three.amount, four.amount);
            indexedGlobalTranches[i] = GlobalTranches(Tranche(a),
Tranche(b), Tranche(c), Tranche(d), Tranche(e));
        }

        $.lastGlobalIndexVoSPOOLMigrated += endIndex * TRANCHES_PER_WORD;
        emit GlobalTranchesMigrated($.lastGlobalIndexVoSPOOLMigrated);
    }
```

YELAY8-9

INCOMPLETE CHECKS IN THE SPOOLDISABLED MODIFIER OF YELAYMIGRATOR

SEVERITY: Medium

PATH:

src/YelayMigrator.sol::_spoolDisabled()#L182-L184

REMEDIATION:

The spoolDisabled modifier should also confirm that ownership has been renounced.

STATUS: Fixed

DESCRIPTION:

The **YelayMigrator.sol** contract uses the **spoolDisabled()** modifier to ensure that migration of balances and stakes can only occur when the SPOOL contract is fully disabled. However, the current implementation only checks whether the contract is paused, neglecting to confirm that ownership has been renounced (i.e., transferred to the zero address). As a result, the owner could still unpause the contract or make modifications, interfering with the migration process.

This is in contradiction with the documentation string of the function.

```
     * @notice Modifier to ensure that the SPOOL contract is disabled
(paused and the owner is the zero address).
     */
    modifier spoolDisabled() {
        _spoolDisabled();
        _;
    }
    /**
     * @notice Ensure that the SPOOL contract is paused and the owner is the
zero address.
     * @dev This ensures that SPOOL migration can only occur when the SPOOL
system is fully disabled.
     */
    function _spoolDisabled() private view {
        require(SPOOL.paused(), "YelayMigrator:_spoolDisabled: SPOOL is
enabled");
    }
```

AN ATTACKER CAN DOS THE TRANSFER OF A USER

SEVERITY: Medium

PATH:

src/sYLAY.sol#L159-160

src/YelayStaking.sol#L102-123

src/sYLAY.sol#L158-183

REMEDIATION:

We recommend adding permission checks to the transferUser function. It can work by checking if the from address has permissions to transfer their account to the to address.

STATUS: Fixed

DESCRIPTION:

In the **YelayStaking.sol** contract the user has the ability to transfer their account to another uninitialized address. That's done by calling the **transferUser()** function which transfers all of the funds from the old address to the new. At the end of the function it calls **sYLAY.transferUser(msg.sender, to)**.

In **sYLAY.sol** contract the **transferUser()** function first checks whether the **from** address is initialized and whether the to address is uninitialized.

```
require(_userGraduals[from].lastUpdatedTrancheIndex != 0, "sYLAY::migrate:
User does not exist");
require(_userGraduals[to].lastUpdatedTrancheIndex == 0, "sYLAY::migrate:
User already exists");
```

However this introduces the following attack vector for a potential attacker.

The attacker can monitor the mempool and when they see that the victim wants to transfer their funds to another address they can simply front-run the transaction and call **transferUser()** where the **to** address is the address that the victim wants to transfer their funds to. After that the victim will be unable to transfer their funds to their new address as it already is initialized so their old funds will still be in their old account. This same cycle can happen indefinitely as the attacker can always create a new account and transfer it to the new address of the victim.

```
function transferUser(address to) external nonReentrant
updateRewards(msg.sender) {
    balances[to] = balances[msg.sender];
    canStakeFor[to] = canStakeFor[msg.sender];
    stakedBy[to] = stakedBy[msg.sender];

    delete balances[msg.sender];
    delete canStakeFor[msg.sender];
    delete stakedBy[msg.sender];

    uint256 rewardTokensCount = rewardTokens.length;
    for (uint256 i; i < rewardTokensCount; i++) {
        RewardConfiguration storage config =
rewardConfiguration[rewardTokens[i]];

        config.rewards[to] = config.rewards[msg.sender];
        config.userRewardPerTokenPaid[to] =
config.userRewardPerTokenPaid[msg.sender];

        delete config.rewards[msg.sender];
        delete config.userRewardPerTokenPaid[msg.sender];
    }

    sYLAY.transferUser(msg.sender, to);
}
```

```
function transferUser(address from, address to) external onlyGradualMinter {
    require(_userGraduals[from].lastUpdatedTrancheIndex != 0,
"sYLAY::migrate: User does not exist");
    require(_userGraduals[to].lastUpdatedTrancheIndex == 0, "sYLAY::migrate:
User already exists");

    UserGradual memory _userGradual = _userGraduals[from];

    // Migrate user tranches
    if (_hasTranches(_userGradual)) {
        uint256 fromIndex = _userGradual.oldestTranchePosition.arrayIndex;
        uint256 toIndex = _userGradual.latestTranchePosition.arrayIndex;

        for (uint256 i = fromIndex; i <= toIndex; i++) {
            userTranches[to][i] = userTranches[from][i];
            delete userTranches[from][i];
        }
    }

    // Migrate user gradual and instant power
    _userGraduals[to] = _userGraduals[from];
    delete _userGraduals[from];

    userInstantPower[to] = userInstantPower[from];
    delete userInstantPower[from];

    emit UserTransferred(from, to);
}
```

YELAY8-17

INCORRECT ROUNDING IN CONVERSIONLIB

SEVERITY: Medium

PATH:

src/libraries/ConversionLib.sol:_applyConversion#L110-L122

REMEDIATION:

The documentation string states that this function should round down in case of exactly half, so the conditional inside the function should be:

```
if (multiplied % 1e18 > half) {
    return (multiplied + half) / 1e18;
} else {
    return multiplied / 1e18;
}
```

STATUS: Fixed

DESCRIPTION:

The function **_applyConversion** in ConversionLib simply applies the SPOOL → YLAY conversion rate of roughly 7.142 to the given amount. Because the rate is a long floating point number, the function also does rounding.

However, the rounding is done incorrectly: if the number should be rounded up, it is rounded down, and if the number should be rounded done, it is lowered by a whole unit.

For example, if the **amount** is 1, then **multiplied** would be **7.142e18**. **multiplied % 1e18 = 0.142e18** and so it is less than **half**, resulting in the second branch being followed. Taking out **half** and doing a division with truncation gives: **6.642e18 / 1e18 = 6**, while the result should be **7**.

Similarly, numbers that should be rounded up, are actually rounded down because of the **/ 1e18** division truncation in the first branch.

```
    function _applyConversion(uint256 amount) private pure returns (uint256)
{
        // Multiply by the conversion rate
        uint256 multiplied = amount * CONVERSION_RATE;

        // Apply rounding: (value + half) / 1e18. If exactly halfway, rounds
down.
        uint256 half = 1e18 / 2;
        if (multiplied % 1e18 >= half) {
            return multiplied / 1e18;
        } else {
            return (multiplied - half) / 1e18;
        }
    }
```

YELAY8-21

STAKING IN SYLAY SHOULD BE DISALLOWED DURING MIGRATION

SEVERITY: Medium

PATH:

src/YelayMigrator.sol:_migrateBalance, _migrateStake#L154-L165

REMEDIATION:

Staking of YLAY into sYLAY should be disallowed until the migration of tranches and stake has been completed. This could be done by pausing YLAY or using a modifier in sYLAYBase that checks the migration status.

STATUS: Fixed

DESCRIPTION:

The YelayMigrator contract has functions to facilitate the SPOOL to YLAY migration. The function **_migrateBalance** allows a user to convert their SPOOL to YLAY immediately after the SPOOL token has been paused and even before the migration has started. The **_migrateStake** function rewards users with YLAY for their pending rewards from SpoolStaking and so they can also acquire YLAY during the migration this way.

Staking into YelayStaking would modify the global and user's graduals and tranches, which can lead to accounting errors if this happens before or during the migration, as the migrate functions all directly overwrite these values.

For example, consider the following scenario:

1. A start state with:
    a. SPOOL is paused;
    b. User A has 1000 SPOOL staked in voSPOOL;
    c. User B has a balance of 1000 SPOOL.

2. User A immediately migrates their 1000 SPOOL to 1000 YLAY.
3. The Yelay team initiates the migration, which sets the **firstTrancheStartTime** of **sYLAY**.
4. User A stakes their 1000 YLAY into sYLAY.
5. The Yelay team migrates the global tranches, which will overwrite the user A's stake for the global gradual and tranche to user B's 1000 voSPOOL.
6. User A unstakes their 1000 YLAY, making the total in the global tranche 0.

```
    function _migrateStake(address staker) private returns (uint256) {
        require(!migratedStake[staker], "YelayMigrator:_migrateStake: Staker
already migrated");
        migratedStake[staker] = true;

        (uint256 yelayStaked, uint256 yelayRewards) =
yelayStaking.migrateUser(staker);

        if (yelayRewards > 0) {
            // Send claimed rewards directly to the user
            YLAY.claim(staker, yelayRewards);
        }

        sYLAY.migrateUser(staker);

        emit StakeMigrated(staker, yelayStaked, yelayRewards);

        return yelayStaked;
    }

    function _migrateBalance(address claimant) private {
        require(!blocklist[claimant], "YelayMigrator:_migrateBalance: User
is blocklisted");
```

```
        require(!migratedBalance[claimant], "YelayMigrator:migrateBalance:
User already migrated");
        migratedBalance[claimant] = true;

        uint256 spoolBalance = SPOOL.balanceOf(claimant);
        uint256 ylayAmount = ConversionLib.convert(spoolBalance);

        YLAY.claim(claimant, ylayAmount);

        emit BalanceMigrated(claimant, spoolBalance, ylayAmount);
    }
```

YELAY8-4

MISSING DISABLEINITIALIZERS() IN CONSTRUCTOR TO PREVENT UNINITIALIZED CONTRACTS

SEVERITY: Informational

PATH:

src/YelayStakingBase.sol#L94-L105

REMEDIATION:

We recommend using the _disableInitializers() function to block initialize calls in the implementation. The _disableInitializers() function was introduced from the OZ v4.6.0, but the project uses

// OpenZeppelin Contracts (last updated v4.5.0) (proxy/utils/Initializable.sol)

Consider using the latest version v5.0.0: openzeppelin-contracts-upgradeable/contracts/proxy/utils/Initializable.sol at master · OpenZeppelin/openzeppelin-contracts-upgradeable

STATUS: Acknowledged

DESCRIPTION:

The **YelayStakingBase.sol** contract uses OpenZeppelin **Initializable** but doesn't call **_disableInitializers** in the constructor per the OpenZeppelin documentation:

Writing Upgradeable Contracts - OpenZeppelin Docs

Proxies - OpenZeppelin Docs

Contract implementations could be initialized when this should not be possible.

```
constructor(
        address _stakingToken,
        address _sYlay,
        address _sYlayRewards,
        address _rewardDistributor,
        address _yelayOwner
    ) YelayOwnable(IYelayOwner(_yelayOwner)) {
        stakingToken = IERC20(_stakingToken);
        sYlay = IsYLAY(_sYlay);
        sYlayRewards = IsYLAYRewards(_sYlayRewards);
        rewardDistributor = IRewardDistributor(_rewardDistributor);
    }
```

YELAY8-5

USE CUSTOM ERRORS

SEVERITY: Informational

REMEDIATION:

Replace require statements with Custom Errors for a more streamlined and user-friendly experience. Furthermore, custom errors are much clearer as they allow for parameter values, making debugging much easier.

STATUS: Acknowledged

DESCRIPTION:

All project errors are currently handled using native Solidity **revert** statements, with some exceptions.

Custom Errors, available from Solidity compiler version 0.8.4, provide benefits such as smaller contract size, improved gas efficiency, and better protocol interoperability. Replace require statements with Custom Errors for a more streamlined and user-friendly experience. Furthermore, custom errors are much clearer as they allow for parameter values, making debugging much easier.

```
require(migrator == _msgSender() || isYelayOwner(), "YLAY::claim: Caller is not the migrator or owner");
```

YELAY8-7

MERGING THE YELAYOWNABLE AND YELAYOWNER CONTRACTS FOR GAS OPTIMIZATION

SEVERITY: Informational

PATH:

src/YelayOwner.sol

src/YelayOwnable.sol

REMEDIATION:

Combine the functionality of YelayOwnable and YelayOwner into a single contract, to reduce the complexity and gas costs.

STATUS: Acknowledged

DESCRIPTION:

In the current implementation, the **Yelay** ownership system is divided into two separate contracts: **YelayOwnable** and **YelayOwner**. This structure introduces inefficiencies in terms of gas consumption, as the **onlyOwner** modifier in **YelayOwnable** relies on an external contract call to the **YelayOwner** contract to verify ownership:

```
    function isYelayOwner() internal view returns (bool) {
        return yelayOwner.isYelayOwner(msg.sender);
    }

    modifier onlyOwner() {
        require(isYelayOwner(), "YelayOwnable::onlyOwner: Caller is not the Yelay
owner");
        _;
    }
```

So every time the **onlyOwner** modifier is used in the **YelayOwnable** contract, a call to **yelayOwner.isYelayOwner(msg.sender)** is made. This external call introduces extra gas costs.

USING CONSOLE IN THE PRODUCTION CODE

SEVERITY: Informational

PATH:

src/libraries/ConversionLib.sol#L4

REMEDIATION:

Consider removing the import.

STATUS: Fixed

DESCRIPTION:

Importing **forge-std/console.sol** in production code is not recommended.

```
import "forge-std/console.sol";
```

YELAY8-13

DISCREPANCIES IN SYLAYBASE DOCS

SEVERITY: Informational

PATH:

src/sYLAYBase.sol#L119

REMEDIATION:

Update the comment.

STATUS: Fixed

DESCRIPTION:

In the **sYLAYBase.sol** contract, the gradual voting power maturation period has been extended from 3 years (156 weeks) to 4 years (208 weeks), as seen in the following code:

```
uint256 public constant FULL_POWER_TRANCHES_COUNT = 52 * 4;
uint256 public constant FULL_POWER_TIME = TRANCHE_TIME *
FULL_POWER_TRANCHES_COUNT;
```

Despite this change in the code, the accompanying comment incorrectly states that the gradual voting power matures over 156 weeks (3 years), which was the behaviour of the **voSPOOL** contract. The comment was likely copied from **voSPOOL** but was not updated to match the code changes.

```
    uint256 public constant FULL_POWER_TRANCHES_COUNT = 52 * 4;
    /// @notice time until gradual power is fully-matured
    /// @dev full power time is 156 weeks (approximately 3 years)
    uint256 public constant FULL_POWER_TIME = TRANCHE_TIME *
FULL_POWER_TRANCHES_COUNT;
```

CONVERTLIB IN-LINE CALCULATIONS WITH LITERALS SHOULD BE REPLACED BY CONSTANTS

SEVERITY: Informational

PATH:

src/libraries/ConversionLib.sol:_applyConversion#L115

REMEDIATION:

Define a constant and replace occurrences of half with HALF:

uint256 private constant HALF = 0.5e18;

STATUS: Fixed

DESCRIPTION:

In the ConversionLib, on line 115 there is an in-line calculations with only literals and this is then assigned to a stack variable:

```
uint256 half = 1e18 / 2;
```

This is a waste of gas, as a constant could be used directly.

```
    function _applyConversion(uint256 amount) private pure returns (uint256)
{
        // Multiply by the conversion rate
        uint256 multiplied = amount * CONVERSION_RATE;

        // Apply rounding: (value + half) / 1e18. If exactly halfway, rounds
down.
        uint256 half = 1e18 / 2;
        if (multiplied % 1e18 >= half) {
            return multiplied / 1e18;
        } else {
            return (multiplied - half) / 1e18;
        }
    }
```

YELAY8-20

PUSH-BASED USER MIGRATION IS VULNERABLE TO SYBIL ATTACK

SEVERITY: Informational

PATH:

src/YelayMigrator.sol:migrateStake#L115-L123

REMEDIATION:

Even though the likelihood is very low, the number of users should be manually verified to be practically possible to migrate, before the migration takes place.

STATUS: Fixed

DESCRIPTION:

The migration from SPOOL to YLAY uses a push-based migration for users' stake in voSPOOL to sYLAY. This means that the internal migration function needs to be executed for every user. The **migrateStake** function itself is not cheap in terms of gas cost.

This approach is vulnerable to a sybil attack, where an attacker could create a lot of users in voSPOOL before the migration, to increase the gas cost and duration of the migration. If the migration were to exceed the duration of a tranche, it could lead to accounting errors in sYLAY.

```
    function migrateStake(address[] calldata claimants) external onlyOwner
spoolDisabled {
        uint256 yelayToStake;
        for (uint256 i = 0; i < claimants.length; i++) {
            yelayToStake += _migrateStake(claimants[i]);
        }

        // Transfer the cumulative staking balance to the yelayStaking
contract
        YLAY.claim(address(yelayStaking), yelayToStake);
    }
```


hexens
×
yelay