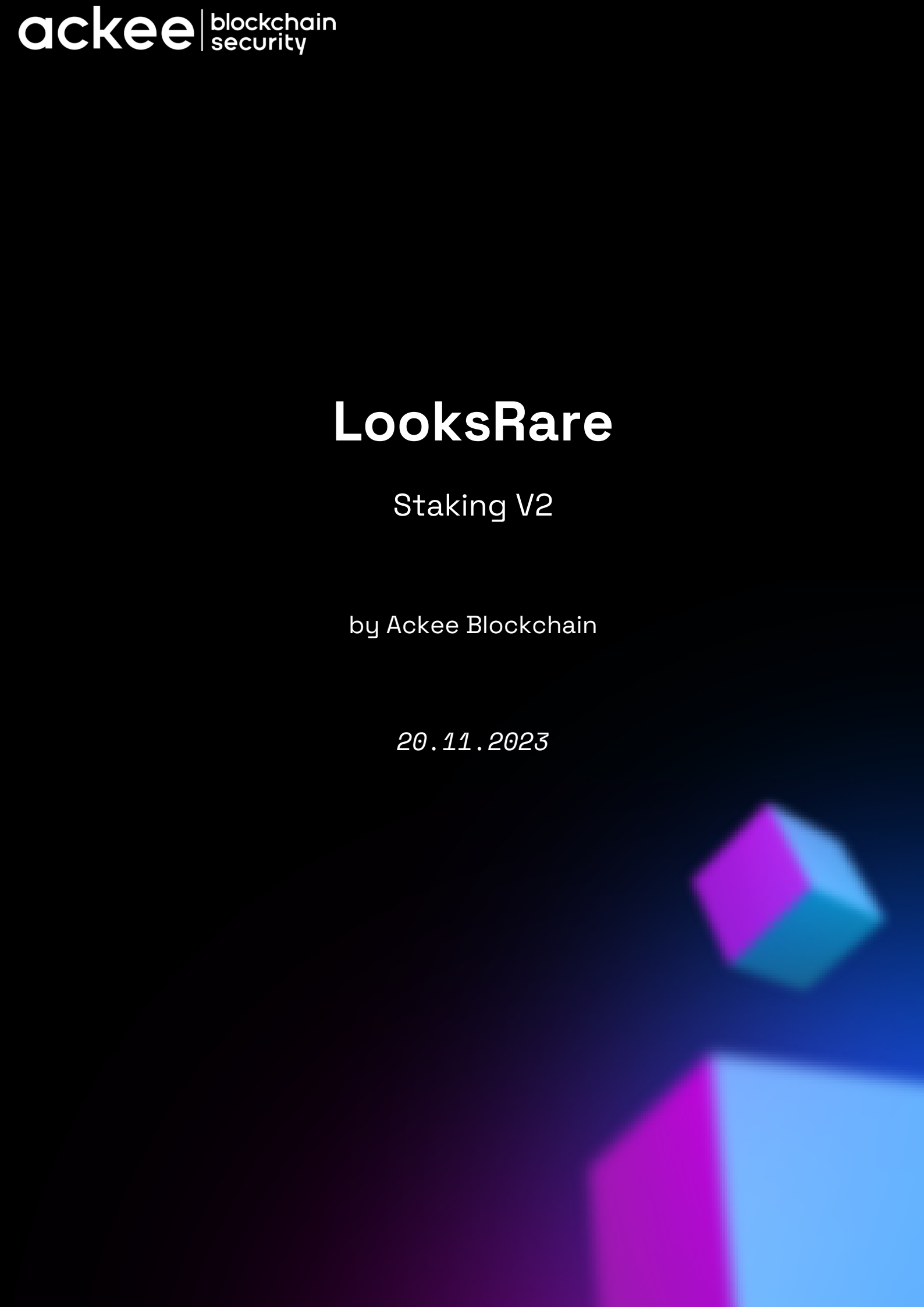
ackee | blockchain security

LooksRare

Staking V2

by Ackee Blockchain

20.11.2023



ackee | blockchain security

Contents

1. Document Revisions

0.1	Draft report	16.11.2023
1.0	Final report	20.11.2023
1.1	Fix review	20.11.2023

2. Overview

This document presents our findings in reviewed contracts.

2.1. Ackee Blockchain

Ackee Blockchain is an auditing company based in Prague, Czech Republic, specializing in audits and security assessments. Our mission is to build a stronger blockchain community by sharing knowledge – we run free certification courses School of Solana, Summer School of Solidity and teach at the Czech Technical University in Prague. Ackee Blockchain is backed by the largest VC fund focused on blockchain and DeFi in Europe, RockawayX.

2.2. Audit Methodology

1. **Technical specification/documentation** - a brief overview of the system is requested from the client and the scope of the audit is defined.
2. **Tool-based analysis** - deep check with automated Solidity analysis tools and Woke is performed.
3. **Manual code review** - the code is checked line by line for common vulnerabilities, code duplication, best practices and the code architecture is reviewed.
4. **Local deployment + hacking** - the contracts are deployed locally and we try to attack the system and break it.
5. **Unit and fuzz testing** - run unit tests to ensure that the system works as expected, potentially write missing unit or fuzz tests.

2.3. Finding classification

A *Severity* rating of each finding is determined as a synthesis of two sub-ratings: *Impact* and *Likelihood*. It ranges from *Informational* to *Critical*.

If we have found a scenario in which an issue is exploitable, it will be assigned an impact rating of *High*, *Medium*, or *Low*, based on the direness of the consequences it has on the system. If we haven't found a way, or the issue is only exploitable given a change in configuration (such as deployment scripts, compiler configuration, use of multi-signature wallets for owners, etc.) or given a change in the codebase, then it will be assigned an impact rating of *Warning* or *Info*.

Low to *High* impact issues also have a *Likelihood*, which measures the probability of exploitability during runtime.

The full definitions are as follows:

Severity

		Likelihood			
		High	**Medium**	**Low**	**-**
Impact	**High**	Critical	High	Medium	-
	Medium	High	Medium	Low	-
	Low	Medium	Low	Low	-
	Warning	-	-	-	Warning
	Info	-	-	-	Info

Table 1. Severity of findings

Impact

- **High** - Code that activates the issue will lead to undefined or catastrophic consequences for the system.
- **Medium** - Code that activates the issue will result in consequences of serious substance.
- **Low** - Code that activates the issue will have outcomes on the system that are either recoverable or don't jeopardize its regular functioning.
- **Warning** - The issue cannot be exploited given the current code and/or configuration (such as deployment scripts, compiler configuration, use of multi-signature wallets for owners, etc.), but could be a security vulnerability if these were to change slightly. If we haven't found a way to exploit the issue given the time constraints, it might be marked as a "Warning" or higher, based on our best estimate of whether it is currently exploitable.
- **Info** - The issue is on the borderline between code quality and security. Examples include insufficient logging for critical operations. Another example is that the issue would be security-related if code or configuration (see above) was to change.

Likelihood

- **High** - The issue is exploitable by virtually anyone under virtually any circumstance.
- **Medium** - Exploiting the issue currently requires non-trivial preconditions.
- **Low** - Exploiting the issue requires strict preconditions.

2.4. Review team

Member's Name	Position
Jan Kalivoda	Lead Auditor
Lukas Bohm	Auditor
Josef Gattermayer, Ph.D.	Audit Supervisor

2.5. Disclaimer

We've put our best effort to find all vulnerabilities in the system, however our findings shouldn't be considered as a complete list of all existing issues. The statements made in this document should not be interpreted as investment or legal advice, nor should its authors be held accountable for decisions made based on them.

3. Executive Summary

LooksRare is a decentralized NFT marketplace. The scope introduces 4 new contracts to the system that allow users to get staking tokens for their LOOKS tokens and accrue rewards by staking them.

Revision 1.0

LooksRare engaged Ackee Blockchain to perform a security review of the LooksRare contracts with a total time donation of 4 engineering days in a period between November 9 and November 15, 2023 and the lead auditor was Jan Kalivoda.

The audit has been performed on the commit `7002474` [1] and the scope was the following:

- StakingRewards.sol
- RewardsDistribution.sol
- AutoCompounder.sol
- StakeableLooksRareToken.sol

We began our review by using static analysis tools, namely Woke. We then took a deep dive into the logic of the contracts. For testing and fuzzing, we have involved Woke testing framework. During the review, we paid special attention to:

- checking the withdrawal requests for staking tokens are processed correctly,
- validating the AutoCompounder contract follows the ERC-4626 standard,
- checking the Uniswap V3 interactions,

- ensuring the reward system can not be abused,
- ensuring the arithmetic of the system is correct,
- detecting possible reentrancies in the code,
- ensuring access controls are not too relaxed or too strict,
- looking for common issues such as data validation.

Our review resulted in 10 findings, ranging from Info to High severity. The most severe one is the possibility of manipulating the reward rate to zero and causing a denial of service (see H1: Anyone can prevent other users from getting rewards).

Ackee Blockchain recommends LooksRare:

- pay more attention to the data validation,
- address all other reported issues.

See Revision 1.0 for the system overview of the codebase.

Revision 1.1

The review was done on the given commit: `5c98408` [2] and the scope were the issues found in the previous revision.

The high severity issue was fixed along with M2, M3. The remaining issues were acknowledged.

See Revision 1.1 for the review of the updated codebase and additional information we consider essential for the current scope.

[1] full commit hash: `7002474b33ab0584150bfe8a72a0cdf6d18dee66`

[2] full commit hash: `5c98408e03744669f46619d0a21a443312c7fd67`

4. Summary of Findings

The following table summarizes the findings we identified during our review. Unless overridden for purposes of readability, each finding contains:

- a *Description*,
- an *Exploit scenario*,
- a *Recommendation* and if applicable
- a *Fix*.

There might often be multiple ways to solve or alleviate the issue, with varying requirements regarding the necessary changes to the codebase. In that case, we will try to enumerate them all, clarifying which solves the underlying issue better (albeit possibly only with architectural changes) than others.

	Severity	Reported	Status
H1: Anyone can prevent other users from getting rewards	High	1.0	Fixed
M1: WETH sent to the staking can be locked forever	Medium	1.0	Acknowledged
M2: Missing data validation for `setNewRewardsDistribution`	Medium	1.0	Fixed
M3: The reward duration can be set as a zero	Medium	1.0	Fixed
L1: Zero-address checks in constructors	Low	1.0	Acknowledged

	Severity	Reported	Status
W1: Slippage overflow on type-casting	Warning	1.0	Acknowledged
W2: Possible overflow on Uniswap interaction	Warning	1.0	Acknowledged
W3: Unlimited allowance	Warning	1.0	Acknowledged
I1: Inconsistent error messages	Info	1.0	Acknowledged
I2: The `withdraw` function has ambiguous errors	Info	1.0	Acknowledged

Table 2. Table of Findings

5. Report revision 1.0

5.1. System Overview

This section contains an outline of the audited contracts. Note that this is meant for understandability purposes and does not replace project documentation.

Contracts

Contracts we find important for better understanding are described in the following section.

StakeableLooksRareToken

The stakeable version of the LOOKS token. This contract allows 3 types of deposits:

- Deposit LOOKS to get minted 1:1 staked LOOKS.
- Deposit LOOKS and stake it into the StakingRewards contract.
- Deposit LOOKS and deposit it into the AutoCompounder contract.

Withdrawals can be handled with a delay or immediately with a 10% penalty (these LOOKS tokens are discarded, decreasing total supply). When the withdrawal request is submitted, then there is 1 week waiting time. Only one request can be active at a time and can be also canceled.

RewardsDistribution

The contract can recover any tokens from StakingRewards and this contract. It can also change the reward distributor address for some different RewardsDistribution contract.

The main function is `transferETH`, which takes this contract's ETH balance and

exchanges it for WETH. This WETH is then sent to the StakingRewards contract and the `notifyRewardAmount` function is called (sets the reward rate).

StakingRewards

The contract uses StakeableLooksRareToken as a staking token and WETH token for rewards. Rewards are accrued based on the active reward rate that is determined by the rewarding period and amount of WETH that is sent into the contract.

AutoCompounder

It is an implementation of ERC-4626. It allows to deposit StakeableLooksRareToken as a vault asset to get minted shares. These shares can be later redeemed for the asset with the accrued rewards. Rewards from staking are automatically swapped in Uniswap V3 Pool (WETH/LOOKS) to LOOKS tokens and these are used to get more StakeableLooksRareToken for restake.

Actors

This part describes actors of the system, their roles, and permissions.

Transfer Manager

Out-of-scope contract handles transfers of LOOKS token.

Staking Owner

Owner of the RewardsDistribution contract. Implemented by LooksRare's `OwnableTwoSteps` contract.

AutoCompounder Owner

Owner of the AutoCompounder contract. Implemented by LooksRare's `OwnableTwoSteps` contract.

Rewards distributor

The contract that can adjust the reward rate for the StakingRewards contract.

5.2. Trust Model

Users have to trust the protocol owner(s) (Staking Owner and AutoCompounder Owner).

The Staking Owner is in charge of the RewardsDistribution contract that is in charge of the StakingRewards contract. This gives him certain privileges:

- Set the RewardsDistribution contract for staking.
- Set the StakingRewards contract for rewards distribution.
- Call the `notifyRewardAmount` function that adjusts the reward rate.

And additionally, recover lost tokens. While this feature can be useful, it also allows to perform (almost) any arbitrary actions for the owner. It can not call `stakingToken` or `rewardToken`, which is a good constraint, but still, for example, it allows some unexpected actions like impersonations of the StakingRewards and RewardsDistribution addresses with the unprotected call.

The AutoCompounder Owner is in charge of the AutoCompounder contract. This gives him the following privileges:

- Set slippage for the users.
- Set minimum swap amount.
- Pause deposits.

Overall, there are some elevated privileges, but also there are several countermeasures to decrease the power of the owners to just necessary actions by design.

H1: Anyone can prevent other users from getting rewards

High severity issue

Impact:	High	Likelihood:	Medium
Target:	StakingRewards, RewardsDistribution	Type:	Access control

Description

In general, the current design allows anyone to manipulate the reward rate.

The `transferETH` function in the RewardsDistribution contract is callable by anyone. It can be called after the rewarding period to set rewards for the next one. The function takes a balance of the contract so if there is no prepared balance prior to the end of the period anyone can call the function (for example with transferring 1 wei to the contract) to set the next reward rate as zero. The reward rate will be set as a zero due to logic in the `notifyRewardAmount` function (e.g. `rewardRate = 1 / 604800`).

```
    function notifyRewardAmount(
        uint256 reward
    ) external override onlyRewardsDistribution nonReentrant
updateReward(address(0)) {
        if (block.timestamp >= periodFinish) {
            // If no reward is currently being distributed, the new rate is
just `reward / duration`
            rewardRate = reward / rewardsDuration;
```

Exploit scenario

The reward period ended and the owner of the protocol wants to start another one so he/she sends ether to the RewardsDistribution contract and

calls the `transferETH` function. Mallory spots this action and front-runs the owner by sending 1 wei and calling the function. As a result, the `notifyRewardAmount` function is called with `reward` parameter equal to 1 and this calculates to a zero reward rate. As a result, users lose their potential rewards.

Recommendation

Add access control for the `transferETH` function since it doesn't need to be callable by anyone by design.

Fix 1.1

Fixed by adding access control to the `transferETH` function.

Go back to Findings Summary

M1: WETH sent to the staking can be locked forever

Medium severity issue

Impact:	Medium	Likelihood:	Medium
Target:	StakingRewards	Type:	Data validation

Description

The RewardsDistribution contract allows setting an arbitrary amount of WETH to the staking contract. However, no mechanism controls what happens when the amount is less than the period. In that case, the reward amount is equal to 0 and the funds are locked in the staking because they are not paid or included in another period.

The only moment when the WETH balance of staking is considered is for the invariant in the `notifyRewardAmount` function.

```
uint256 balance = rewardToken.balanceOf(address(this));
require(rewardRate <= balance / rewardsDuration, "91");
```

Moreover, over time how rewards are paid off, there are some WETH leftovers (not significant amounts).

Exploit scenario

The reward period is set to 1 week. The `notifyRewardAmount` function is called with the amount of 604 800 - 1. As a result, 604 799 wei of WETH are locked forever.

Recommendation

Add data validation to prevent sending less value than the reward period.

Consider design adjustments to prevent this locking behavior.

Go back to Findings Summary

M2: Missing data validation for `setNewRewardsDistribution`

Medium severity issue

Impact:	High	Likelihood:	Low
Target:	StakingRewards	Type:	Data validation

Description

The `setNewRewardsDistribution` function is missing any data validation.

Exploit scenario

By accident, an incorrect value is passed to the setter function. Instead of reverting, the call succeeds.

Recommendation

At least, add zero-address check. Ideally, each component that is passed to another should have a constant variable that contains the contract id that can be used for validation.

For example, the RewardsDistribution contract will contain a variable named `CONTRACT_ID`:

```
bytes32 public constant CONTRACT_ID = keccak256("Looksrare: Rewards
Distributor");
```

and the `setNewRewardsDistribution` function in StakingRewards will contain a check for the value of this variable:

```
require(
    RewardsDistribution(_rewardsDistribution).CONTRACT_ID() == keccak256
("Looksrare: Rewards Distributor"),
```

```
    "Invalid reward distributor address"
);
```

This will help to reduce the risk of passing incorrect values.

Fix 1.1

Fixed similarly to the recommendation above. The `setNewRewardsDistribution` function now contains a function:

```
function isRewardsDistribution() external pure returns (bool) {
    return true;
}
```

That is being called in the new function implementation and thus, the check is performed.

```
function setNewRewardsDistribution(address rewardsDistribution) external
onlyRole(DEFAULT_ADMIN_ROLE) {
    if (IRewardsDistribution(rewardsDistribution).isRewardsDistribution())
{
        stakingRewards.setNewRewardsDistribution(rewardsDistribution);
        emit StakingRewardsRewardsDistributionUpdated(rewardsDistribution);
    }
}
```

Go back to Findings Summary

M3: The reward duration can be set as a zero

Medium severity issue

Impact:	High	Likelihood:	Low
Target:	StakingRewards	Type:	Data validation

Description

In the constructor is the `rewardsDuration` variable set and it is the only place where it can be set. There is no data validation for this variable.

Also, this variable is used for divisions in the `notifyRewardAmount` function so setting this variable to zero will lead to permanent DoS of this function.

```
    function notifyRewardAmount(
        uint256 reward
    ) external override onlyRewardsDistribution nonReentrant
updateReward(address(0)) {
        if (block.timestamp >= periodFinish) {
            // If no reward is currently being distributed, the new rate is
just `reward / duration`
            rewardRate = reward / rewardsDuration;
```

Exploit scenario

By accident, a zero value is passed and it is not discovered right after the deployment. Users stake some tokens and the protocol starts functioning, however, when rewards are transferred to the contract, it fails on division by zero.

Recommendation

At least add a check against a zero value. Ideally, set a range of acceptable values.

Fix 1.1

Fixed by adding `require` statement to the constructor for a non-zero value.

Go back to Findings Summary

L1: Zero-address checks in constructors

Low severity issue

Impact:	Medium	Likelihood:	Low
Target:	StakeableLooksRareToken, RewardsDistribution, AutoCompounder	Type:	Data validation

Description

The contracts are missing zero-address checks for input parameters that are passing addresses in their constructors. Passing zero-address can cause the need for redeployment due to a non-functional state. Or even worse, it can introduce more severe problems, like loss of funds, if it is not discovered right after the deployment.

Exploit scenario

By accident, an incorrect value is passed to the constructor. Instead of reverting, the call succeeds.

Recommendation

Add zero-address checks for all input parameters that are passing addresses (like StakingRewards already has).

Go back to Findings Summary

W1: Slippage overflow on type-casting

Impact:	Warning	Likelihood:	N/A
Target:	AutoCompounder	Type:	Integer overflow

Description

The `slippageBp` variable is susceptible to integer overflow, but since there are defined allowed ranges, it doesn't introduce an issue in the current scope.

```
function setSlippage(uint256 _slippageBp) external onlyOwner {
    if (_slippageBp < MIN_SLIPPAGE_BP || _slippageBp > MAX_SLIPPAGE_BP) {
        revert AutoCompounder__InvalidSlippageBp();
    }
    slippageBp = uint16(_slippageBp);

    emit SlippageBpUpdated(_slippageBp);
}
```

However, if the slippage limits will be adjusted in a future development, then there is a risk of overflow. For example, calling `setSlippage(65538)` will result into `slippageBp == 2`.

Recommendation

Apply safe type-casting with over/underflow checks.

Go back to Findings Summary

W2: Possible overflow on Uniswap interaction

Impact:	Warning	Likelihood:	N/A
Target:	AutoCompounder	Type:	Integer overflow

Description

The `_sellRewardTokenForAssetIfAny` function is called for almost any action. This function calls also the `_sellRewardAmountOutMinimum` function that takes uint256 parameter. However, in the body of the function, there is performed unsafe type-casting that can potentially result in integer overflow.

```
    function _sellRewardAmountOutMinimum(uint256 amountIn) private view
returns (uint256 amountOutMinimum) {
        (int24 arithmeticMeanTick, ) = OracleLibrary.consult({pool:
uniswapV3Pool, secondsAgo: 600});
        uint256 quote = OracleLibrary.getQuoteAtTick({
            tick: arithmeticMeanTick,
            baseAmount: uint128(amountIn),
            baseToken: address(rewardToken),
            quoteToken: LOOKS
        });
```

The likelihood that this happen is very low, but still possible for some tokens since we will need for example 10**20 tokens with 18 decimals.

Recommendation

Consider the possibility of integer overflow with the tokens that are going to be used and the gas efficiency of the solution that is going to prevent this. If it will remain unprotected for the sake of gas, add a warning to the inlined documentation.

Go back to Findings Summary

W3: Unlimited allowance

Impact:	Warning	Likelihood:	N/A
Target:	AutoCompounder	Type:	Access control

Description

In the constructor, there is granted unlimited allowance to Uniswap and StakingRewards.

```
_rewardToken.approve(_uniswapRouter, type(uint256).max);
_stakingToken.approve(_stakingRewards, type(uint256).max);
```

This is gas-efficient and convenient, however, presents a risk of losing these assets when the addresses get compromised.

Recommendation

Add approvals only when needed for a needed amount.

Go back to Findings Summary

I1: Inconsistent error messages

Impact:	Info	Likelihood:	N/A
Target:	`**/*`	Type:	Logging

Description

The StakingRewards contract contains ambiguous error messages (numbers), e.g.

```
require(_stakingToken != _rewardToken, "69");
```

and the rest of the codebase uses custom errors. Such errors do not contain enough information to easily parse why the given transaction failed.

Recommendation

Stick to custom errors and use them consistently accross the entire codebase.

Go back to Findings Summary

I2: The `withdraw` function has ambiguous errors

Impact:	Info	Likelihood:	N/A
Target:	StakingRewards	Type:	Logging

Description

The `withdraw` contract doesn't have any error handling, instead of it, it relies on reverts on under/overflow.

```
_totalSupply = _totalSupply - amount;
_balances[msg.sender] = _balances[msg.sender] - amount;
```

Such errors do not contain enough information to easily parse why the given transaction failed.

Recommendation

Add proper custom errors to have always well defined state.

Go back to Findings Summary

6. Report revision 1.1

- The StakeableLooksRareToken was renamed to WrappedLooksRareToken and its symbol sLOOKS to wLOOKS.
- The WrappedLooksRareToken deposit/withdraw functions were renamed to wrap/unwrap.
- The WrappedLooksRareToken is now limited to be transferred into the contract itself.
- The instant withdrawal fee was adjusted from 10% to 5%.
- The AutoCompounder token xLOOKS was renamed to cLOOKS.
- The AutoCompounder token cLOOKS is now untransferable.
- The maximum slippage (`MAX_SLIPPAGE_BP`) was increased from 10% to 100%, so now the owner has more flexibility in setting the slippage for the autocompounding.
- The RewardsDistribution contract is now using OpenZeppelin's AccessControl contract instead of the Ownable contract from LooksRare.

Appendix A: How to cite

Please cite this document as:

Ackee Blockchain, LooksRare: Staking V2, 20.11.2023.

Appendix B: Glossary of terms

The following terms might be used throughout the document:

Superclass/Ancestor of C

A contract that C inherits/derives from.

Subclass/Child of C

A contract that inherits/derives from C.

Syntactic contract

A Solidity contract. May have an inheritance chain, and may be deployed.

Deployed contract

An EVM account with non-zero code. If its source was written in Solidity, it was created through at least one syntactic contract. If that contract had superclasses (parents), it would be composed of multiple syntactic contracts.

Init/initialization function

A non-constructor function that serves as an initializer. Often used in upgradeable contracts.

External entrypoint

A `public` or `external` function.

Public/Publicly-accessible function/entrypoint

An `external` or `public` function that can be successfully executed by any network account.

Mutating function

A non-`view` and non-`pure` function.

ackee | blockchain security

Thank You

Ackee Blockchain a.s.


Prague, Czech Republic


hello@ackeeblockchain.com


https://twitter.com/AckeeBlockchain