CANTINA

Uniswap UniStaker

Security Review

Cantina Managed review by:

Xmxanuel, Lead Security Researcher

High Byte, Security Researcher

Jeiwan, Security Researcher

April 25, 2024

Contents

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

Cantina Managed provides a detailed evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While Cantina Managed endeavors to identify and disclose all potential security issues, it cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities that were absent during the initial review. Therefore, any changes made to the code require a new security review to ensure that the code remains secure. Please be advised that the Cantina Managed security review is not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity	Description
Critical	*Must* fix as soon as possible (if already deployed).
High	Leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
Medium	Global losses <10% or losses to only a subset of users, but still unacceptable.
Low	Losses will be annoying but bearable. Applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

Uniswap is an open source decentralized exchange that facilitates automated transactions between ERC20 token tokens on various EVM-based chains through the use of liquidity pools and automatic market makers (AMM).

From Apr 1st to Apr 4th the Cantina team conducted a review of UniStaker on commit hash 2ac42a50. The team identified a total of **12** issues in the following risk categories:

- Critical Risk: 0
- High Risk: 0
- Medium Risk: 0
- Low Risk: 4
- Gas Optimizations: 3
- Informational: 5

3 Findings

3.1 Low Risk

3.1.1 Calling `UniStaker.claimReward` multiple times reduces the overall received reward amount

Severity: Low Risk

Context: UniStaker.sol#L805

Description: The `UniStaker` contract utilizes a higher precision of `1e36` to track the accumulated rewards of a beneficiary as well as for `scaledRewardRate`. This allows stream and accumulated fractions of the smallest unit per second.

Such precision is particularly beneficial for tokens with lower precisions, like USDC (10^6) because the `rewardRate` is expressed in per second.

In the `_claimReward` function the rewards are converted back to `REWARD_TOKEN` precision by dividing them by the `SCALE_FACTOR`. As a result, fractional amounts, such as half of a `usdcWei`, are truncated since they cannot be paid out in the precision of the reward token.

However, afterwards the `scaledUnclaimedRewardCheckpoint` is set to zero. Which means the accumulated fractions of the smallest unit are lost for the `_beneficiary`. This leads to the effect, that calling the `claimReward` multiple times over a period of time can impact the final amount of tokens received.

Calling the `claimReward` multiple times should not impact the reward amount.

Example: For instance, if a `beneficiary` is supposed to receive `100 USDC` plus half a `usdcWei` after `15 days`, the total after `30 days` would amount to `200 USDC` plus one usdcWei (equivalent to `200.000001 USDC`). A call of the `claimReward` after 15 days would cancel the accumulated 1/2 usdcWei and the final amount would be only `200 USDC`:

```
uint256 _reward = scaledUnclaimedRewardCheckpoint[_beneficiary] / SCALE_FACTOR;
if (_reward == 0) return;
scaledUnclaimedRewardCheckpoint[_beneficiary] = 0;
```

Recommendation: Instead of setting it to zero keep the accumulated fractions in the checkpoint.

```
uint256 _reward = scaledUnclaimedRewardCheckpoint[_beneficiary] / SCALE_FACTOR;
if (_reward == 0) return;
scaledUnclaimedRewardCheckpoint[_beneficiary] = scaledUnclaimedRewardCheckpoint[_beneficiary]  - (_reward *
↪  SCALE_FACTOR);
```

Uniswap Foundation: Fixed in commit bac90312.

Cantina Managed: Fixed.

3.1.2 `STAKEN_TOKEN.permit` can be front-run in a signature griefing attack

Severity: Low Risk

Context: UniStaker.sol#L307

Description: There is an existing known vulnerability related to `ERC-2612` permit calls inside of a contract.

Someone could front-run the `STAKEN_TOKEN.permit` transaction call with the signature from the mempool.

```
STAKE_TOKEN.permit(msg.sender, address(this), _amount, _deadline, _v, _r, _s);
_depositId = _stake(msg.sender, _amount, _delegatee, _beneficiary);
```

This would lead to a revert inside the contract because the permit has already happened and the `nonce` is increased. See this source for a more detailed explanation.

There are two affected function in the `UniStaker` contract:

- UniStaker.permitAndStake
- UniStaker.permitAndStakeMore

Recommendation: Adding a try/catch block would ignore potential revert in the permit function:

```
try STAKE_TOKEN.permit(msg.sender, address(this), _amount, _deadline, _v, _r, _s) {} catch {};
```

If the approval is missing the transaction would anyway revert in the `safeTransferFrom`.

Uniswap Foundation: Fixed in commit 3c8f649c.

Cantina Managed: Fixed.

3.1.3 Rewards notified when there are no staked tokens cannot be claimed

Severity: Low Risk

Context: UniStaker.sol#L609

Description: The UniStaker.notifyRewardAmount() function triggers the distribution of a newly transferred reward among stakers. The function recomputes the rate at which the reward is distributed, however, the distribution only happens when some amount of tokens are staked (UniStaker.sol#L235-L238):

```
function rewardPerTokenAccumulated() public view returns (uint256) {
  if (totalStaked == 0) return rewardPerTokenAccumulatedCheckpoint;

  return rewardPerTokenAccumulatedCheckpoint
    + (scaledRewardRate * (lastTimeRewardDistributed() - lastCheckpointTime)) / totalStaked;
}
```

In the above function, the `rewardPerTokenAccumulatedCheckpoint` variable is increased only when `totalStaked` is positive. This means that no rewards will be earned when there are no staked tokens since rewards distribution is spread over time. As a result, since the checkpoint timestamp will still be updated during the next rewards checkpointing (UniStaker.sol#L812-L815), the reward distributed over the period when there were not staked tokens cannot be claimed by stakers. Also, it cannot be rolled over by a subsequent call to `notifyRewardAmount()`.

The described situation can happen in two cases:

1. After the deployment of the contract: if `UniStaker.notifyRewardAmount()` is called before any tokens are staked.

2. At any point of time: if all tokens were unstaked (e.g. as a result of a higher incentive in another staking protocol) but rewards are kept notified.

Recommendation: Short term, ensure that the `UniStaker` contract is deployed and enabled, and some tokens are staked before the first rewards are notified. Long term, consider adding a mechanism of accounting of rewards notified during the periods of 0 staked tokens and redistributing these rewards among future stakers.

Uniswap Foundation: Acknowledged. Rewards distributed when no one was staking should logically remain unclaimed, aligning with the principle that if no one stakes, no one earns the rewards. Practically speaking, the odds of this happening are zero.

Cantina Managed: Acknowledged.

3.1.4 A caller of `V3FactoryOwner.claimFees` can not define expected `payoutAmount` for a safety check

Severity: Low Risk

Context: V3FactoryOwner.sol#L183

Description: Anyone can claim the available protocol fees from a Uniswap pool by calling `claimFees` on the `V3FactoryOwner`. In exchange, the caller needs to pay a predefined `payoutAmount` to receive the entire accumulated fees of both tokens in a pool.

The caller can specify the expected amounts of tokens (`_amount0Requested` and `_amount1Requested`) they wish to receive but not the amount of tokens they need to pay (`payoutAmount`). The `payoutAmount` is fixed and stored in the contract.

However, `payoutAmount` can be changed by the `admin`. For Uniswap, the `admin` is the Timelock contract, which is part of their governance.

This scenario can lead to an edge case where the `payoutAmount` is increased while a `claimFee` transaction, created under the assumption of a different `payoutAmount`, is still in the `mempool`. (This assumes that the caller has also granted a higher `PAYOUT_TOKEN` approval).

The caller would pay more as expected in this scenario for the fee token.

In most cases, an MEV searcher will detect a fee trading opportunity as soon as it becomes profitable. This issue does not apply here.

Recommendation: Consider adding an `_expectedPayoutAmount` parameter to guarantee the expected `payoutAmount` in all scenarios otherwise revert.

```
function claimFees(
  IUniswapV3PoolOwnerActions _pool,
  address _recipient,
  uint128 _amount0Requested,
  uint128 _amount1Requested,
  uint256 _expectedPayoutAmount,
) external returns (uint128, uint128) {
 if(_expectedPayoutAmount != payoutAmount) {
   revert V3FactoryOwner__UnexpectedPayoutAmount();
 }
```

Uniswap Foundation: Acknowledged. A `payoutAmount` change would involve a governance process includling a timelock. The assumption is that callers of this method are aware of potential changes and should consider them. A comment for clarification has been added in commit bac90312.

Cantina Managed: Resolved.

3.2 Gas Optimization

3.2.1 Using `CREATE2` for `surrogates` deployment would remove the need to store them

Severity: Gas Optimization

Context: UniStaker.sol#L664

Description: The `_fetchOrDeploySurrogate` function deploys a new `DelegationSurrogate` contract if not existing and stores the address in the `surrogates` mapping.

Recommendation: Using `CREATE2` together with the `_delegatee` address as `salt` would remove the need to store the `surrogates` address in storage. The OpenZeppelin library CREATE2 could be used.

The `_fetchOrDeploySurrogate` function would first compute the address by calling `Create2.computeAddress` using the `delegatee` as `salt`. Afterwards the `extcodesize` op-code can check if the contract exists. If not a new deployment should happen with `create2` and the delegatee as `salt`.

Uniswap Foundation: Acknowledged. We'll stick with the simpler deployment method to keep complexity low despite the gas difference.

Cantina Managed: Acknowledged.

3.2.2 Deposit balance type can be reduced to `uint96`

Severity: Gas Optimization

Context: UniStaker.sol#L98, UNI token

Description: The `balance` field of the Deposit structure uses type `uint256`. The field stores the amount of UNI tokens deposited to the contract in a separate deposit (UniStaker.sol#L714). However, the UNI token contract stores token balances in `uint96` variables (UNI token), thus using `uint256` for storing deposited amounts is excessive.

Recommendation: Consider using type `uint96` to store deposit amounts to enable slot packing and optimize gas usage of deposit storage operations.

Uniswap Foundation: Fixed in commit 31d49232.

Cantina Managed: Fixed.

3.2.3 Invalid reward rate calculation can be simplified

Severity: Gas Optimization

Context: UniStaker.sol#L626

Description: An invalid reward rate is smaller than UniStaker.SCALE_FACTOR (it's not possible to distribute less than 1 unit of reward token per second). To detect such a rate, the quotient of `scaledRewardRate / SCALE_FACTOR` is compared to 0:

```
if ((scaledRewardRate / SCALE_FACTOR) == 0) revert UniStaker__InvalidRewardRate();
```

However, `scaledRewardRate` can be directly compared to `SCALE_FACTOR` to reduce the gas cost of the computation:

```
if (scaledRewardRate < SCALE_FACTOR) revert UniStaker__InvalidRewardRate();
```

Recommendation: Consider simplifying the invalid reward rate check.

Uniswap Foundation: Acknowledged. We'll leave as-is for code readability and gas optimization is extremely minimal.

Cantina Managed: Acknowledged.

3.3 Informational

3.3.1 `SignatureChecker` can call using ERC1271 `isValidSignature`

Severity: Informational

Context: UniStaker.sol#L871

Description: `SignatureChecker` uses ERC1271 which performs external `staticcall`. This could lead to reentrancy although there are no negative implications and limited to read-only context.

Recommendation: If ERC1271 is not expected to be used support for it can be removed, for gas savings and future proofing security.

Uniswap Foundation: Acknowledged. We want 1271 functionality so we will not make a change here.

Cantina Managed: Acknowledged.

3.3.2 Use two-step ownership transfer

Severity: Informational

Context: UniStaker.sol#L208

Description: Currently, the `setAdmin` function simply changes the admin address.

Recommendation: It is best practice to use two-step transfer such as Ownable2Step.sol.

Uniswap Foundation: Acknowledged. Because ownership will be held by governance, in practice, the 2 step process does not add much. We will not implement it.

Cantina Managed: Acknowledged.

3.3.3 `UniStaker.claimReward` should return the claimed amount

Severity: Informational

Context: UniStaker.sol#L569

Description: The `claimReward` function does not return the claimed amount. As a result, a third-party protocol integrating this function would need to calculate the difference between the balances before and after the call.

Recommendation: Add a return value to indicate the amount claimed.

Uniswap Foundation: Fixed in commit 927cb11f7.

Cantina Managed: Fixed.

3.3.4 The `V3FactoryOwner` has no passthrough function for `Factory.setOwner`

Severity: Informational

Context: V3FactoryOwner.sol#L131

Description: In the current Uniswap Governance design the Uniswap Timelock contract is the owner of the `UniswapV3Factory` which can deploy new Uniswap pools. After the change, the new `owner` will be the `V3FactoryOwner` contract. The `V3FactoryOwner` will be controlled by the Timelock contract.

The `V3FactoryOwner` does not have a passthrough method needed for Governance to call `UniswapV3Factory.setOwner` again, this means after the ownership of the `UniswapV3Factory` is transferred to the `V3FactoryOwner` it will be permanent and can not be reversed.

Recommendation: The not reversible change of the ownership to `V3FactoryOwner` should be documented.

Uniswap Foundation: Acknowledged. This is the desired behavior. No change. We may add some documentation to `V3FactoryOwner` natspec to make this choice more explicit.

Cantina Managed: Acknowledged.

3.3.5 Inaccurate code documentation

Severity: Informational

Context: UniStaker.sol

Description: In the following cases, code documentation doesn't match the actual implementation:

- `_deadline` argument specifies the timestamp *after* which the signature expires, not *at* which: UniStaker.sol#L291, UniStaker.sol#L318, UniStaker.sol#L370, UniStaker.sol#L396, UniStaker.sol#L440, UniStaker.sol#L490, UniStaker.sol#L539, UniStaker.sol#L576, UniStaker.sol#L858.
- `permitAndStake()` and `permitAndStakeMore()` don't require token approvals since they make them via signed permits: UniStaker.sol#L284-L285, UniStaker.sol#L365-L366. Also, the usage of *at least* in "to spend at least the would-be staked amount" is not accurate because it's not possible to approve more tokens than is to be staked (`_amount` is always approved and staked).

Recommendation: Consider improving code documentation in the highlighted cases.

Uniswap Foundation: Fixed in commit fcdce769.

Cantina Managed: Fixed.