Interpol Security Review

Pashov Audit Group

Conducted by: sashik-eth, krikoeth, pontifex, saksham

December 24th 2024 - January 2nd 2024

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of multiple teams of some of the best smart contract security researchers in the space. Having a combined reported security vulnerabilities count of over 1000, the group strives to create the absolute very best audit journey possible - although 100% security can never be guaranteed, we do guarantee the best efforts of our experienced researchers for your blockchain protocol. Check our previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

3. Introduction

A time-boxed security review of the **0xHoneyJar/interpol** repository was done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

4. About Interpol

InterPol on Berachain allows users to lock their liquidity while still earning rewards. The HoneyLocker contract enables users to deposit and lock LP tokens for specified durations and stake them in vaults through secure adapters. The HoneyQueen manages approved adapters, tracks reward tokens, and calculates fees. This structure helps liquidity providers retain full control of their assets while participating in staking rewards.

5. Risk Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

5.1. Impact

- High - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- Medium - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- Low - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

5.2. Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

5.3. Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

6. Security Assessment Summary

review commit hash - 1a7dd55a13f4a1c5d604f8f5ad2b7a63202243a6

fixes review commit hash - c1d91a0be07aff5e997bc9fea368b952bfafc109

Scope

The following smart contracts were in scope of the audit:

- `LockerFactory`
- `HoneyQueen`
- `HoneyLocker`
- `Beekeeper`
- `AdapterFactory`
- `TokenReceiver`
- `BGTStationAdapter`
- `BaseVaultAdapter`
- `BeradromeAdapter`
- `InfraredAdapter`
- `KodiakAdapter`

7. Executive Summary

Over the course of the security review, sashik-eth, krikoeth, pontifex, saksham engaged with HoneyJar to review Interpol. In this period of time a total of **15** issues were uncovered.

Protocol Summary

Protocol Name	Interpol
Repository	https://github.com/0xHoneyJar/interpol
Date	December 24th 2024 - January 2nd 2024
Protocol Type	Protocol-owned liquidity management

Findings Count

Severity	**Amount**
High	1
Medium	3
Low	11
Total Findings	**15**

Summary of Findings

ID	Title	Severity	Status
[H-01]	Reward tokens can be withdrawn without fee	High	Resolved
[M-01]	Rewards might be temporarily stuck in Kodiak adapter	Medium	Resolved
[M-02]	Attacker can DoS lockers from unstaking from Kodiak	Medium	Resolved
[M-03]	ERC20 tokens without return value will DoS reward claiming	Medium	Resolved
[L-01]	The locker owner can redirect all fees to the referrer	Low	Acknowledged
[L-02]	Updating fee share when an overriding referrer exists	Low	Resolved
[L-03]	Locker does not have onERC721Received hook	Low	Resolved
[L-04]	expirations is set to 0 for non-unlocked tokens	Low	Resolved
[L-05]	Kodiak vault can revert unstaking/claiming calls	Low	Acknowledged
[L-06]	_hasSetOperator is not set to true	Low	Resolved
[L-07]	Inheriting Initializable and disabling initializers	Low	Resolved
[L-08]	Blocked tokens can be deposited but cannot be withdrawn	Low	Resolved
[L-09]	The expirations can always be less than block.timestamp	Low	Acknowledged

[L-10]	Some adapters might be unable to stake ERC721 tokens	Low	Resolved
[L-11]	Kodiak staking time is not adjustable	Low	Resolved

8. Findings

8.1. High Findings

[H-01] Reward tokens can be withdrawn without fee

Severity

Impact: Medium

Likelihood: High

Description

InterPol takes a cut of the rewards accrued from staking of the tokens locked in the protocol when withdrawing the reward tokens. The rewards can be withdrawn with the `withdrawERC20` function, which also sends the fee to the protocol. However, users can use the `withdrawLPToken` function to bypass the fee. This happens due to the lines checking if the withdrawn token is a reward token being commented out, as seen in the snippet:

```
function withdrawLPToken(
       address_LPToken,
       uint256_amountOrId
    ) external onlyUnblockedTokens(_LPToken
         // @audit-issue These lines should not be commented
         // if (HONEY_QUEEN.isRewardToken(_LPToken)) revert HasToBeLPToken();
         //if (expirations[_LPToken] == 0) revert HoneyLocker__HasToBeLPToken();
         if (block.timestamp < expirations[_LPToken])
              revert HoneyLocker__NotExpiredYet();

         // self approval only needed for ERC20, try/catch in case it's an ERC721
         try ERC721(_LPToken).approve(address(this), _amountOrId) {} catch {}
         ERC721(_LPToken).transferFrom(address(this), recipient(), _amountOrId);
         emit HoneyLocker__Withdrawn(_LPToken, _amountOrId);
    }
```

Recommendations

Consider uncommenting the two lines that check whether the token being withdrawn is a reward token.

Keep in mind that currently, if depositing to an unlocked locker, the expiration is set to zero. This means that upon uncommenting of the second line (`//if (expirations[_LPToken] == 0) revert HoneyLocker__HasToBeLPToken();`), it will be impossible to withdraw tokens deposited to an unlocked locker without a fee.

However, this line seems to be redundant, as if a token is not registered as a reward token, users can lock 1 wei of the token to avoid fee payment.

8.2. Medium Findings

[M-01] Rewards might be temporarily stuck in Kodiak adapter

Severity

Impact: Medium

Likelihood: Medium

Description

When unstaking tokens from Kodiak, the rewards are claimed, as we can observe in the code snippet below:

```
function withdrawLocked
      (bytes32 kek_id) nonReentrant withdrawalsNotPaused public {
        _withdrawLocked(msg.sender, msg.sender, kek_id, true);
    }

    function _withdrawLocked(
      addressstaker_address,
      addressdestination_address,
      bytes32kek_id,
      boolcollectRewards
    ) internal  {
        // Collect rewards first and then update the balances
        if(collectRewards) {
            _getReward(staker_address, destination_address);
        }

        //... snippet ...

    }
```

This means that upon unstaking, the rewards are sent to the adapter contract. Since the unstake function does not send these out, this breaks the protocol invariant, that besides special occasions, the adapter should not hold any funds. The invariant can be broken for a short period, until the rewards are claimed and sent out from the adapter. However, sometimes the reward collection is paused, which will make the `getReward` function call revert:

```
function getReward() external nonReentrant returns (uint256[] memory) {
        require(rewardsCollectionPaused == false,"Rewards collection paused");
        return _getReward(msg.sender, msg.sender);
    }
```

This means that calling `claim` on Kodiak adapter would revert as well, making the rewards stuck in the adapter until the reward collection is unpaused in Kodiak. While it is understandable that claiming from Kodiak does not work if reward collection is paused, this prevents rewards already present in the Adapter contract from being redeemed in the locker contract.

The impact is considered medium as users' rewards would be temporarily locked in the adapter contract and a protocol invariant is broken. The likelihood is medium since reward collection pausing is a legitimate feature of Kodiak that could be activated at any time for various reasons.

Recommendations

Consider wrapping the `getReward` function in try-catch block. This way the call will always be able to send rewards out from the contract.

```
function claim(address vault) external override onlyLocker isVaultValid
      (vault) returns (address[] memory, uint256[] memory) {
        IKodiakFarm kodiakFarm = IKodiakFarm(vault);

        address[] memory rewardTokens = kodiakFarm.getAllRewardTokens();
        uint256[] memory amounts = new uint256[](rewardTokens.length);
-       kodiakFarm.getReward();
+       try kodiakFarm.getReward() {} catch {}
        for (uint256 i; i < rewardTokens.length; i++) {
            amounts[i] = ERC20(rewardTokens[i]).balanceOf(address(this));
            /*
                we skip the transfer, to not block any other rewards
                it can always be retrieved later because we use the balanceOf
                  () function
            */
            try ERC20(rewardTokens[i]).transfer(locker, amounts[i]) {} catch {
                emit Adapter__FailedTransfer
                  (locker, rewardTokens[i], amounts[i]);
                amounts[i] = 0;
            }
        }
        return (rewardTokens, amounts);
    }
```

Another thing to consider is sending rewards to the locker upon unstaking so the invariant that the adapter should not hold assets would always hold.

[M-02] Attacker can DoS lockers from unstaking from Kodiak

Severity

Impact: Medium

Likelihood: Medium

Description

When staking in the `HoneyLocker` contract, the fields `totalLPStaked` and `vaultLPStaked` are increased by the amount of tokens being staked. When unstaking, these fields are decreased by the amount of tokens being unstaked. However, an attacker can exploit this mechanism by donating one wei of the token to the adapter, leading to arithmetic underflow when unstaking from the Kodiak adapter. This would cause the unstake function to always fail and make the staked tokens unwithdrawable. The following unit test proves the issue:

```
function test_unstakeDoS(
        bool _useOperator
    ) external prankAsTHJ(_useOperator) {
        // too low amount results in the withdrawal failing because of how
        // Kodiak works
        uint256 amountToDeposit = 1000e18;

        StdCheats.deal(address(LP_TOKEN), address(locker), amountToDeposit);

        bytes32 expectedKekId = keccak256(
            abi.encodePacked(
                address(lockerAdapter),
                block.timestamp,
                amountToDeposit,
                GAUGE.lockedLiquidityOf(address(lockerAdapter))
            )
        );

        locker.stake(address(GAUGE), amountToDeposit);

        vm.warp(block.timestamp + 30 days);
        GAUGE.sync();

        StdCheats.deal(address(LP_TOKEN), address(lockerAdapter), 1);

        vm.expectRevert();
        locker.unstake(address(GAUGE), uint256(expectedKekId));
    }
```

After running the test, it will pass, meaning the `unstake` call has reverted after donating 1 wei of `LP_TOKEN` to the adapter contract.

The medium impact rating is justified because if exploited in a live contract, the issue could be fixed by upgrading the adapter or locker to a new version, which would make the funds accessible. However, this would still mean a temporary freeze.

It can also be fixed by staking the donated amount by the locker. While this would mean lost funds for the locker, the attacker is not incentivized to spend their funds on these donations. Due to the combination of these factors, the likelihood should be deemed medium.

Recommendations

This issue occurs due to the unique mechanism of staking in Kodiak (compared to other adapters). The best approach would be to track stake amounts internally in the Kodiak adapter contract. The proposed fix could look like the following:

```
+   mapping(bytes32 kekId => uint256 amount) amounts;

    function stake(
      addressvault,
      uint256amount
    ) external override onlyLocker isVaultValid(vault
        IKodiakFarm kodiakFarm = IKodiakFarm(vault);
        address token = kodiakFarm.stakingToken();

        ERC20(token).transferFrom(locker, address(this), amount);
        ERC20(token).approve(address(kodiakFarm), amount);
+       bytes32 expectedKekId = keccak256(
+           abi.encodePacked(
+               address(this),
+               block.timestamp,
+               amount,
+               kodiakFarm.lockedLiquidityOf(address(this))
+           )
+       );
        kodiakFarm.stakeLocked(amount, kodiakFarm.lock_time_for_max_multiplier
          ());
+       amounts[expectedKekId] = amount;
        return amount;
    }

    function unstake(
      addressvault,
      uint256kekIdAsUint
    ) external override onlyLocker isVaultValid(vault
        IKodiakFarm kodiakFarm = IKodiakFarm(vault);
        address token = kodiakFarm.stakingToken();

        kodiakFarm.withdrawLocked(bytes32(kekIdAsUint));
+       uint256 unstakeAmount = amounts[bytes32(kekIdAsUint)];
        uint256 amount = ERC20(token).balanceOf(address(this));
        ERC20(token).transfer(locker, amount);
+       delete amounts[bytes32(kekIdAsUint)];
-       return amount;
+       return unstakeAmount;
    }
```

However, due to the upgradeable nature of the adapters, the proposed fix should also be combined with adding a `gap` field to the `BaseVaultAdapter` contract (see more here).

[M-03] ERC20 tokens without return value will DoS reward claiming

Severity

Impact: Medium

Likelihood: Medium

Description

Several functions throughout the application use the `transfer` function to transfer ERC20 tokens. However, some tokens do not return a `bool` on transfer, and since the ERC20 interface expects the `bool` return value, calling `transfer` on tokens that do not return a `bool` would revert. The following functions suffer from this issue:

- `withdrawERC20` in `HoneyLocker`
- `unstake` in `BeradromeAdapter`
- `unstake` in `BGTStationAdapter`
- `unstake` in `InfraredAdapter`
- `unstake` in `KodiakAdapter`

The following functions also call `transfer` on ERC20, but the call is wrapped in a try-catch block. However, such a call would still revert:

- `claim` in `BeradromeAdapter`
- `claim` in `InfraredAdapter`
- `claim` in `KodiakAdapter`

The impact is rated as medium since the issue can be resolved through a contract upgrade, though users would face a temporary freeze of their assets until the fix is deployed. The likelihood is high given that numerous widely-used tokens in the ecosystem don't strictly follow the ERC20 standard regarding return values.

Recommendations

Consider using the `SafeTransfer` library for transferring ERC20 tokens. This can still make some transfers revert the whole transaction, therefore consider adding a function to pull tokens out separately.

Alternatively, consider implementing the safe transfer functionality in the adapters on your own and making it not revert on failure, so the transfer can be retried later.

8.3. Low Findings

[L-01] The locker owner can redirect all fees to the referrer

The `Beekeeper` contract distributes fees from reward tokens to the protocol and the referrer, if present. However, the fee share for the referrer is calculated with the `mulDivUp` function, which rounds up, see the snippet.

```
function distributeFees
      (address _referrer, address _token, uint256 _amount) external payable {
        // snippet //

        uint256 referrerFee = FPML.mulDivUp
          (_amount, referrerFeeShareInBps, 10000);

        // snippet //
    }
```

This means that a user can intentionally withdraw rewards in such a way that the total fee amounts to 1 wei, which would make the fee for the referrer 1 wei due to rounding up, resulting in the protocol receiving no cut.

Although there is no incentive for users to behave in this way since they would probably overpay on fees, there might be an instance where this could make sense, for example with WBTC rewards due to high value per 1 wei, but the griefer would not gain anything unless they are also the referrer, in which case it could save them the fees. This scenario is highly unlikely.

While it is recommended to round in favor of the protocol, this could have the opposite impact where the referrer could receive no fee share. Therefore, the best recommendation would be to add a check for minimum reward amount claimed.

[L-02] Updating fee share when an overriding referrer exists

This is how the referrer fee is assigned in the BeeKeeper function ->

```
function setReferrerFeeShare
  (address _referrer, uint256 _shareOfFeeInBps) external onlyOwner {
        if (!isReferrer[_referrer]) revert Beekeeper__NotAReferrer();
        _referrerFeeShare[_referrer] = _shareOfFeeInBps;
    }
```

But it is possible that the above referrer has an overriding referrer, in that case, the fee share for the overriding referrer should also be updated.

If `referrerOverrides[_referrer]` is non zero then update the overriding referrer fee share.

[L-03] Locker does not have `onERC721Received` hook

The HoneyLocker.sol contract is intended to receive NFT (including ERC1155) tokens , though normally it can receive NFT tokens via a normal ERC721 transfer if the external integration integrates with the locker and leverages `safeTransferFrom` / `safeTransfer` to transfer NFT tokens then since the locker has no onERC721Received (and ERC1155 hooks) hooks the transfer would always revert.

It is recommended to include these hooks to solve the same.

[L-04] `expirations` is set to 0 for non-unlocked tokens

`expirations` of the deposited LP tokens would be set to 0 for unlocked HoneyLocker:

```
function depositAndLock(
      address _LPToken,
      uint256 _amountOrId,
      uint256 _expiration
    ) external onlyOwnerOrOperator {

        if
          (!unlocked && expirations[_LPToken] != 0 && _expiration < expirations[_LPTok
            revert HoneyLocker__ExpirationNotMatching();
        }

        expirations[_LPToken] = unlocked ? 0 : _expiration; <=@
```

This would create a potential footgun for users since they would be able to withdraw LPs using the `withdrawERC20()` function and pay fees for LP tokens when shouldn't.

Consider updating `expirations` to be set to 1 for unlocked HoneyLocker.

[L-05] Kodiak vault can revert unstaking/claiming calls

In some edge cases, `Kodiak` vault could revert unstaking/claiming calls if dust rewards were accrued or dust LP were staked due to rounding small amounts, for example, it would revert if unstaking only 33 wei:

```
function test_dustStake() external prankAsTHJ(true) {
        uint256 amountToDeposit = 33;

        StdCheats.deal(address(LP_TOKEN), address(locker), amountToDeposit);

        //====== Stake

        bytes32 expectedKekId1 = keccak256(
            abi.encodePacked(
                address(
                  lockerAdapter
                ), block.timestamp, amountToDeposit, GAUGE.lockedLiquidityOf(address(l
            )
        );

        locker.stake(address(GAUGE), amountToDeposit);

        vm.warp(block.timestamp + 30 days);

        //====== Unstake

        locker.unstake(address(GAUGE), uint256(expectedKekId1));
    }
```

Fix for this issue is not available inside Interpol itself, but such behavior of the underlying protocol should be considered since it could lead to potential issues in the future when reward rates would be updated and reverting could be caused on non-dust amounts.

[L-06] `_hasSetOperator` is not set to true

`BGTStationAdapter` contract uses `_hasSetOperator` mapping to track if `locker` already set operator for it. Its value is set to true in the

`BGTStationAdapter#stake()` function. But it's not set to true in the `BGTStationAdapter#wildcard()` function. Consider updating it to prevent `locker` from setting the operator for `BGTStationAdapter` multiple times.

[L-07] Inheriting `Initializable` and disabling initializers

All adapter contracts and `HoneyLocker` contract are proxies and should inherit an `Initializable` contract. Their `initialize()` functions should have an `initializer` modifier. Also inside the `constructor` of these contracts `_disableInitializers()` should be called. This would prevent malicious users from calling the `initialize()` function on the implementation contracts.

[L-08] Blocked tokens can be deposited but cannot be withdrawn

The `HoneyLocker.depositAndLock` function does not prevent depositing blocked tokens. So users can deposit such tokens but can't withdraw them. Tokens will be blocked at the contract.

```
>>  function depositAndLock(
  address_LPToken,
  uint256_amountOrId,
  uint256_expiration
) external onlyOwnerOrOperator {
<...>
    function withdrawLPToken(address _LPToken, uint256 _amountOrId) external
>>  onlyUnblockedTokens(_LPToken)
    onlyOwnerOrOperator
```

Consider adding the `onlyUnblockedTokens` modifier in the `depositAndLock` function.

[L-09] The `expirations` can always be less than `block.timestamp`

The `HoneyLocker.depositAndLock` function does not check if the `_expiration` is less than the current timestamp. This way users can always set such `expirations` that let immediate `withdrawLPToken` execution.

```
function depositAndLock(
      address_LPToken,
      uint256_amountOrId,
      uint256_expiration
    ) external onlyOwnerOrOperator {

>>      if
  (!unlocked && expirations[_LPToken] != 0 && _expiration < expirations[_LPToken]) {
            revert HoneyLocker__ExpirationNotMatching();
        }

        expirations[_LPToken] = unlocked ? 0 : _expiration;

        // using transferFrom from ERC721 because same signature for ERC20
        // with the difference that ERC721 doesn't expect a return value
        ERC721(_LPToken).transferFrom(msg.sender, address(this), _amountOrId);

        emit HoneyLocker__Deposited(_LPToken, _amountOrId);
        emit HoneyLocker__LockedUntil(_LPToken, _expiration);
    }
```

Consider checking if the `_expiration` parameter exceeds the `block.timestamp` value.

[L-10] Some adapters might be unable to stake ERC721 tokens

The protocol expects LP tokens to be ERC721 tokens. Since the core feature of the protocol is staking of locked tokens, this should also hold for ERC721 tokens. However, some adapters might be unable to stake ERC721 tokens due to mismatched `approve` function return values.

The signature of `ERC721::approve` is:

```
function approve(address to, uint256 tokenId) external;
```

While the signature of `ERC20::approve` is:

```
function approve(address spender, uint256 value) external returns (bool);
```

This means that calling `approve` via the ERC20 interface on an ERC721 token will revert, as the ERC20 approve expects a return value. This will make the `stake` functions of the `BeradromeAdapter` and the `KodiakAdapter` unavailable for ERC721 tokens.

Consider using the `safeApprove` function from `SafeTransferLib` for approvals. This approach will work with both ERC721 and ERC20 tokens.

[L-11] Kodiak staking time is not adjustable

When staking tokens to Kodiak, the staker chooses how long they want to stake their tokens, and the staking multiplier is then based on this duration. When calling the `stake` function on the Kodiak adapter, the staking time is set to `lock_time_for_max_multiplier()`, which is the maximum possible time.

This can be problematic, as this value is currently `2592000` seconds (30 days). Although the LP tokens are usually locked for longer than 30 days, we should not make this assumption, especially in the case where tokens are locked in an `unlocked` locker.

Another problem might occur if this value changes in the future, for example to one year. If a user wanted to lock their tokens for one month, they would not be able to stake them, as the stake time would be longer than the lock time.

Consider giving the user the option to choose the lock time, for example by utilizing the wildcard function. The code snippet below shows how this could work:

```
+   uint256 lockTime;

    function stake(
      addressvault,
      uint256amount
    ) external override onlyLocker isVaultValid(vault
        IKodiakFarm kodiakFarm = IKodiakFarm(vault);
        address token = kodiakFarm.stakingToken();

        ERC20(token).transferFrom(locker, address(this), amount);
        ERC20(token).approve(address(kodiakFarm), amount);
-       kodiakFarm.stakeLocked(amount, kodiakFarm.lock_time_for_max_multiplier
- ());
+       kodiakFarm.stakeLocked(amount, lockTime);
        return amount;
    }

    function wildcard(
        address vault,
        uint8 func,
        bytes calldata args
    ) external override onlyLocker isVaultValid(vault) {
        IKodiakFarm kodiakFarm = IKodiakFarm(vault);
        XKDK xKdk = XKDK(kodiakFarm.xKdk());
        if (func == 0) {
            (uint256 amount, uint256 duration) = abi.decode(
                args,
                (uint256, uint256)
            );
            xKdk.redeem(amount, duration);
        } else if (func == 1) {
            uint256 index = abi.decode(args, (uint256));
            xKdk.finalizeRedeem(index);
            ERC20(kodiakFarm.kdk()).transfer(locker, ERC20(kodiakFarm.kdk
              ()).balanceOf(address(this)));
+       } else if (func == 2) {
+           lockTime = abi.decode(args, (uint256));
        }
    }
```