Interpol Security Review

Pashov Audit Group

Conducted by: juancito, santipu, peanuts

September 2nd - September 4th

Contents

1. About Pashov Audit Group 3
2. Disclaimer 3
3. Introduction 3
4. About Interpol 3
5. Risk Classification 4
   5.1. Impact 4
   5.2. Likelihood 4
   5.3. Action required for severity levels 5
6. Security Assessment Summary 5
7. Executive Summary 6
8. Findings 9
   8.1. Critical Findings 9
      [C-01] Adversary can lock tokens forever for any HoneyLocker 9
   8.2. High Findings 11
      [H-01] Blocked ERC20 tokens can still be withdrawn 11
      [H-02] BGT and reward tokens can be withdrawn without paying fees 12
      [H-03] All locked LP tokens can be withdrawn during a migration 13
   8.3. Medium Findings 16
      [M-01] Function isTargetContractAllowed always returns true 16
      [M-02] HoneyLocker.unstake() can possibly lead to underflow 17
      [M-03] Users can avoid paying fees to the protocol due to rounding 19
   8.4. Low Findings 21
      [L-01] Missing code check in Solady safeTransfer functions 21
      [L-02] Inconsistent transfer of LP tokens 21
      [L-03] Recommended use of ERC721 safeTransferFrom 22

[L-04] Empty calls can be made to any of the allowed contracts 23
[L-05] HoneyQueen emits wrong old values in events 24
[L-06] Calldata arguments are not checked on stake/unstake 25
[L-07] Approvals should be revoked when unstaking in HoneyLocker.sol 26
[L-08] Returning the fee share for an account that is not a referrer 27
[L-09] The amount of tokens staked per contract can be artificially inflated 27
[L-10] Expiration timing can be set to before the current time 29

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 is a protocol allowing users to lock their liquidity, no matter the duration, without having to renounce to the rewards possible.

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 - dafe92ea1ad300d566a65743bfed4dbdd8e96427

fixes review commit hash - 149c27ba331c4945a8e78cb1b4c2d66dc3470dc1

Scope

The following smart contracts were in scope of the audit:

- `HoneyLocker`
- `HoneyQueen`
- `TokenReceiver`
- `SetAndForgetFactory`
- `Beekeper`
- `LockerFactory`

7. Executive Summary

Over the course of the security review, juancito, santipu, peanuts engaged with HoneyJar to review Interpol. In this period of time a total of **17** issues were uncovered.

Protocol Summary

Protocol Name	Interpol
Repository	https://github.com/0xHoneyJar/interpol
Date	September 2nd - September 4th
Protocol Type	LP locking

Findings Count

Severity	**Amount**
Critical	1
High	3
Medium	3
Low	10
Total Findings	**17**

Summary of Findings

ID	Title	Severity	Status
[C-01]	Adversary can lock tokens forever for any HoneyLocker	Critical	Resolved
[H-01]	Blocked ERC20 tokens can still be withdrawn	High	Resolved
[H-02]	BGT and reward tokens can be withdrawn without paying fees	High	Resolved
[H-03]	All locked LP tokens can be withdrawn during a migration	High	Resolved
[M-01]	Function isTargetContractAllowed always returns true	Medium	Resolved
[M-02]	HoneyLocker.unstake() can possibly lead to underflow	Medium	Resolved
[M-03]	Users can avoid paying fees to the protocol due to rounding	Medium	Resolved
[L-01]	Missing code check in Solady safeTransfer functions	Low	Resolved
[L-02]	Inconsistent transfer of LP tokens	Low	Resolved
[L-03]	Recommended use of ERC721 safeTransferFrom	Low	Resolved
[L-04]	Empty calls can be made to any of the allowed contracts	Low	Resolved
[L-05]	HoneyQueen emits wrong old values in events	Low	Resolved
[L-06]	Calldata arguments are not checked on stake/unstake	Low	Acknowledged

[L-07]	Approvals should be revoked when unstaking in HoneyLocker.sol	Low	Resolved
[L-08]	Returning the fee share for an account that is not a referrer	Low	Acknowledged
[L-09]	The amount of tokens staked per contract can be artificially inflated	Low	Resolved
[L-10]	Expiration timing can be set to before the current time	Low	Acknowledged

8. Findings

8.1. Critical Findings

[C-01] Adversary can lock tokens forever for any HoneyLocker

Severity

Impact: High

Likelihood: High

Description

The `onlyOwnerOrMigratingVault` modifier on `HoneyLocker` has a vulnerable check `owner() != Ownable(msg.sender).owner()`.

The problem with this is that an adversary can create a malicious contract implementing an `owner()` function that returns the current owner of the HoneyLocker and bypasses this check.

```
modifier onlyOwnerOrMigratingVault() {
    if (msg.sender != owner() && owner() != Ownable(msg.sender).owner
      ()) revert Unauthorized();
    _;
}
```

This modifier is used to protect the `depositAndLock()` function. So, an attacker can call it setting the biggest possible `_expiration` value, making it impossible for the legit owner to withdraw their tokens as they will never expire.

Proof of Concept

This test shows how an attacker can modify any expiration and set it to its maximum value.

The victim won't be able to withdraw their tokens later, as they will never expire.

- Copy the `ExpirationAttacker` contract to `test/HoneyLocker.t.sol`
- Copy the `test_depositExpirationAttack` test to `test/HoneyLocker.t.sol` inside the `HoneyLockerTest` test
- Run `forge test --mt test_depositExpirationAttack`

```
contract ExpirationAttacker {
    address immutable victim;

    constructor(address _victim) {
        victim = _victim;
    }

    function owner() external returns (address) {
        return victim;
    }
}
```

```
function test_depositExpirationAttack() external prankAsTHJ {
    ExpirationAttacker attacker = new ExpirationAttacker(THJ);

    vm.stopPrank();
    vm.startPrank(address(attacker));

    assertEq(honeyLocker.expirations(address(HONEYBERA_LP)), 0);

    uint256 maxExpiration = type(uint256).max;
    honeyLocker.depositAndLock(address(HONEYBERA_LP), 0, maxExpiration);

    assertEq(honeyLocker.expirations(address(HONEYBERA_LP)), maxExpiration);
}
```

Recommendations

One possible solution is to add a function for the owner of the `HoneyLocker` to set a specific "Migration Vault" that can call `depositAndLock()`, and update the `onlyOwnerOrMigratingVault()` modifier accordingly.

8.2. High Findings

[H-01] Blocked ERC20 tokens can still be withdrawn

Severity

Impact: Medium

Likelihood: High

Description

Some ERC20 tokens are expected to be blocked in the contract with the `onlyUnblockedTokens` modifier, and the owner shouldn't be able to withdraw them. That's the reason this check is implemented for `withdrawERC20()`:

```
function withdrawERC20(
  address _token,
  uint256 _amount
) external onlyUnblockedTokens(_token

modifier onlyUnblockedTokens(address _token) {
    if (!unlocked && HONEY_QUEEN.isTokenBlocked(_token)) revert TokenBlocked();
    _;
}
```

The issue lies in that it is possible to withdraw those ERC20 tokens via the `withdrawLPToken()` function:

```
function withdrawLPToken(address _LPToken, uint256 _amount) external onlyOwner {
    if (expirations[_LPToken] == 0) revert HasToBeLPToken();
    if (block.timestamp < expirations[_LPToken]) revert NotExpiredYet();

    ERC20(_LPToken).transfer(msg.sender, _amount);
}
```

The function implements a `expirations[_LPToken] == 0` check to prevent non-LP tokens from being withdrawn, but this expectation can be broken by calling `depositAndLock()` with the token, a zero-amount value, and a past expiration:

```
function depositAndLock(
  address _LPToken,
  uint256 _amountOrId,
  uint256 _expiration
) external onlyOwnerOrMigratingVault {
    if
      (!unlocked && expirations[_LPToken] != 0 && _expiration < expirations[_LPToken])
        revert ExpirationNotMatching();
    }
    expirations[_LPToken] = unlocked ? 1 : _expiration;

    ERC721(_LPToken).transferFrom(msg.sender, address(this), _amountOrId);
}
```

This way, the contract owner can set a non-zero past `expirations[_LPToken]` value. This will allow them to withdraw blocked tokens.

Recommendations

Implement the `onlyUnblockedTokens` modifier in either the `withdrawLPToken()` or the `depositAndLock()`, depending if the intention is to allow to deposit and lock such tokens but not withdraw them, or prevent them from being deposited and locked altogether.

[H-02] BGT and reward tokens can be withdrawn without paying fees

Severity

Impact: Medium

Likelihood: High

Description

Some tokens are expected to be held by the `HoneyLocker` contract and are expected to pay fees when withdrawn. Such is the case of the BGT token, and other reward tokens.

To do that, a fee is implemented on the `withdrawBERA()` and `withdrawERC20()` tokens.

The issue lies in that it is possible to withdraw those ERC20 tokens via the `withdrawLPToken()` function:

```
function withdrawLPToken(address _LPToken, uint256 _amount) external onlyOwner {
    if (expirations[_LPToken] == 0) revert HasToBeLPToken();
    if (block.timestamp < expirations[_LPToken]) revert NotExpiredYet();

    ERC20(_LPToken).transfer(msg.sender, _amount);
}
```

The function implements a `expirations[_LPToken] == 0` check to prevent non-LP tokens from being withdrawn, but this expectation can be broken by calling `depositAndLock()` with the token, a zero-amount value, and a past expiration:

```
function depositAndLock(
  address_LPToken,
  uint256_amountOrId,
  uint256_expiration
) external onlyOwnerOrMigratingVault {
    if
      (!unlocked && expirations[_LPToken] != 0 && _expiration < expirations[_LPToken])
        revert ExpirationNotMatching();
    }
    expirations[_LPToken] = unlocked ? 1 : _expiration;

    ERC721(_LPToken).transferFrom(msg.sender, address(this), _amountOrId);
}
```

This way, the contract owner can set a non-zero past `expirations[_LPToken]` value. This will allow them to withdraw tokens without paying fees via `withdrawLPToken()`.

Recommendations

The `expirations[_LPToken] == 0` check in `withdrawLPToken()` is not sufficient to prevent non-LP tokens from being withdrawn with it.

Checking that the token is not the BGT token, works for it. But another whitelist check would be needed to prevent withdrawing non-LP tokens, or effectively only allowing LP-tokens to be deposited via `depositAndLock()`. Any of those options should work.

[H-03] All locked LP tokens can be withdrawn during a migration

Severity

Impact: High

Likelihood: Medium

Description

HoneyLocker migrations are authorized based on the codehashes of the contracts:

```
function migrate(
  address[]calldata_LPTokens,
  uint256[]calldata_amountsOrIds,
  addresspayable_newHoneyLocker
) external onlyOwner {
    // check migration is authorized based on codehashes
    if (!HONEY_QUEEN.isMigrationEnabled(address
      (this).codehash, _newHoneyLocker.codehash)) {
        revert MigrationNotEnabled();
    }

    ...
}
```

The problem is that the function assumes that it would be migrated to another HoneyLocker with the same initialized values.

But it is possible to create a new `HoneyLocker` with different values for `unlocked`, `HONEY_QUEEN`, and `referral`.

This can lead to many impacts. The most notable one is the possibility to withdraw all the LP tokens that were supposed to remain locked until the expiration time.

Other attacks can be performed given a malicious HoneyQueen contract provided by the user.

Proof of Concept

This test shows how some locked tokens can be withdrawn by migrating to an unlocked HoneyLocker.

- Copy the test to `test/HoneyLocker.t.sol`
- Run `forge test --mt test_migrationExploit`

```
function test_migrationExploit() external prankAsTHJ {
    // LEGIT SETUP

    // deposit first some into contract
    uint256 balance = HONEYBERA_LP.balanceOf(THJ);
    HONEYBERA_LP.approve(address(honeyLocker), balance);
    honeyLocker.depositAndLock(address(HONEYBERA_LP), balance, expiration);

    // clone it
    HoneyLockerV2 honeyLockerV2 = new HoneyLockerV2();
    honeyLockerV2.initialize(THJ, address(honeyQueen), referral, false);

    // set hashcode in honeyqueen then attempt migration
    honeyQueen.setMigrationFlag(true, address(honeyLocker).codehash, address
      (honeyLockerV2).codehash);

    // CREATE A NEW LOCKER, INITIALIZE IT WITH MALICIOUS DATA AND MIGRATE TO IT
    HoneyLockerV2 maliciousLocker = new HoneyLockerV2();
    address maliciousHoneyQueen = makeAddr("MALICIOUS_HONEY_QUEEN");
    address anotherReferral = makeAddr("ANOTHER_REFERRAL");
    bool unlocked = true;
    maliciousLocker.initialize(THJ, address
      (maliciousHoneyQueen), anotherReferral, unlocked);

    honeyLocker.migrate(SLA.addresses(address(HONEYBERA_LP)), SLA.uint256s
      (balance), payable(address(maliciousLocker)));
    assertEq(HONEYBERA_LP.balanceOf(address(maliciousLocker)), balance);

    // The malicious owner got back their tokens
    maliciousLocker.withdrawLPToken(address(HONEYBERA_LP), balance);
    assertEq(HONEYBERA_LP.balanceOf(THJ), balance);
}
```

Recommendations

One possible way to mitigate this is to add additional checks to verify that the `unlocked`, `HONEY_QUEEN`, and `referral` values are the same in both locks.

8.3. Medium Findings

[M-01] Function `isTargetContractAllowed` always returns true

Severity

Impact: High

Likelihood: Low

Description

The function `isTargetContractAllowed` is flawed and will always return true due to how memory strings are stored in the EVM.

```
function isTargetContractAllowed(address _target) public view returns
      (bool allowed) {
        string memory protocol = protocolOfTarget[_target];
        assembly {
            allowed := not(iszero(protocol))
        }
    }
```

Here is a code snippet that can be pasted in Chisel as a PoC:

```
➜ string memory protocol;
➜ bool allowed;
➜ assembly { allowed := not(iszero(protocol)) }
```

Now we can check that the value of `allowed` will always be true, even if the string is empty:

```
➜ allowed
Type: bool
└ Value: true
```

The root cause of this issue is how strings are stored in memory in the EVM. The function uses inline assembly to check if the string is empty by evaluating the memory pointer. However, in Solidity, a memory pointer to a string is always non-zero once allocated, regardless of the string's content. Therefore,

the condition `not(iszero(protocol))` in the assembly code incorrectly returns true for every case, even when the string is empty.

Currently, the impact of this issue is minimal because the `isSelectorAllowedForTarget` function effectively prevents the `HoneyLocker` from interacting with unauthorized contracts. However, if the flawed function is utilized in future implementations, it could enable locker owners to fully circumvent the liquidity lock and avoid paying fees.

Recommendations

To fix this issue is recommended to use a different method to check if a string is empty or not:

```
function isTargetContractAllowed(address _target) public view returns
      (bool allowed) {
        string memory protocol = protocolOfTarget[_target];
-       assembly {
-           allowed := not(iszero(protocol))
-       }
+       allowed = bytes(protocol).length > 0
    }
```

And we can use Chisel to make sure the fix works correctly:

```
➜ string memory protocol;
➜ bool allowed = bytes(protocol).length > 0;
➜ allowed
Type: bool
└ Value: false
```

[M-02] `HoneyLocker.unstake()` can possibly lead to underflow

Severity

Impact: High

Likelihood: Low

Description

In the test files, HoneyLocker interacts with the `stakeLocked()` and `withdrawLockedAll()` functions in the Kodiak staking contract.

In the Kodiak staking contract, if `withdrawLockedAll()` is called but the expiration date is not up, the function will simply call `get_rewards()` and skip the `_withdrawLocked()`.

```
function withdrawLockedAll() nonReentrant withdrawalsNotPaused public {
>       _getReward(msg.sender, msg.sender);
        LockedStake[] memory locks = lockedStakes[msg.sender];
        for(uint256 i = 0; i < locks.length; i++) {
>           if
  (locks[i].liquidity > 0 && block.timestamp >= locks[i].ending_timestamp){
                _withdrawLocked(msg.sender, msg.sender, locks[i].kek_id, false);
            }
        }
    }
```

In the HoneyLocker.unstake function, the `staked[_LPToken][_stakingContract]` value will be subtracted.

```
function unstake(
    address _LPToken,
    address _stakingContract,
    uint256 _amount,
    bytes memory _data
  )
        public
        onlyOwner
        onlyAllowedTargetContract(_stakingContract)
        onlyAllowedSelector(_stakingContract, "unstake", _data)
    {
>       staked[_LPToken][_stakingContract] -= _amount;
        (bool success,) = _stakingContract.call(_data);
        if (!success) revert UnstakeFailed();

        emit Unstaked(_stakingContract, _LPToken, _amount);
    }
```

Let's say the owner stakes 100 LP tokens for 30 days, and `staked[_LPToken][_stakingContract]` is 100e18.

- The 100 LP tokens is now inside Kodiak staking.
- 15 days in, the owner calls `unstake()` with `withdrawLockedAll()` as the function selector.
- The function will collect KDK rewards, but the LP tokens will not be sent back to the HoneyLocker as the expiration is not up.
- In the `HoneyLocker.unstake()` function, `staked[_LPToken][_stakingContract]` will now be 0 but LP tokens is still in Kodiak contract.
- 15 days later, the owner calls `unstake()` again to withdraw the LP tokens.
- The `unstake()` function will revert with underflow because it attempts to deduct from `staked[_LPToken][_stakingContract]`.

The 100 LP tokens will be stuck in the Kodiak staking contract.

Recommendations

```
-   staked[_LPToken][_stakingContract] -= _amount;
+   if (_amount >= staked[_LPToken][_stakingContract]) {
+       staked[_LPToken][_stakingContract] -= _amount;
+   }
```

[M-03] Users can avoid paying fees to the protocol due to rounding

Severity

Impact: Medium

Likelihood: Medium

Description

When the owner of a `HoneyLocker` initiates a withdrawal of funds through the `withdrawERC20` or `withdrawBERA` methods, a fee is required to be paid to both the referral and the treasury. However, the owner can take advantage of the rounding down in the EVM to avoid paying any fees at all. The root cause of the issue is in `HoneyQueen::computeFees`:

```
uint256 public fees = 200; // in bps
// ...
function computeFees(uint256 amount) public view returns (uint256) {
    return (amount * fees) / 10000;
}
```

To exploit this, the owner can withdraw funds in very small batches. For instance, if an owner wants to withdraw 1,000,000 tokens, they can repeatedly request a withdrawal of just 49 tokens. This specific amount is chosen because, with a nominal fee rate of 2%, the calculated fee should round down to zero:

```
49 * 200 / 10000 = 0
```

This attack is generally not cost-effective for most tokens because the gas fees required can exceed the benefit. However, it could be viable for high-value

tokens that have fewer decimal places, where the loss from rounding becomes more significant relative to transaction costs.

Recommendations

To mitigate this issue, two solutions are suggested:

1. Modify the fee calculation in `computeFees` to always round up, ensuring that no withdrawal is completely free of charge.
2. Set a minimum fee threshold so that no calculated fee results in zero, irrespective of the withdrawal amount.

8.4. Low Findings

[L-01] Missing code check in Solady safeTransfer functions

The `distributeFees()` function in `Beekeper` uses the `safeTransfer` extension from Solady.

The problem with that is that it doesn't check for the token code size, and it doesn't revert for calls to EOAs.

This means that it would be possible to simulate a token distribution without actually transfering any tokens.

This can be done for a yet-to-exist token that the adversary knows the address. Then they can `distributeFees()` and emit fake `FeesDistributed` events for valid future token.

Recommendation

Check the code size of the token before distributing it.

[L-02] Inconsistent transfer of LP tokens

The `depositAndLock()` function in `HoneyLocker` uses the ERC721 interface as it doesn't expect a return value, and won't revert if the used token doesn't return any.

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

        // 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);

        ...
    }
```

On the other hand `withdrawLPToken()` uses the ERC20 `transfer()` interface:

```
function withdrawLPToken(address _LPToken, uint256 _amount) external onlyOwner {
    ...
    ERC20(_LPToken).transfer(msg.sender, _amount);
}
```

So, in case an LP token doesn't return any value on its `transferFrom()` function, it would be able to be deposited, but won't be able to be withdrawn.

Recommendation

It would be recommended to use the `safeTransfer` and `safeTransferFrom` extensions for the `depositAndLock()` and `withdrawLPToken()` functions for consistency, and to prevent any possibility of locked tokens.

Consider doing the same for `withdrawERC20()`, so that no other possible received ERC20 tokens get locked.

Be mindful that if using `SafeTransferLib` from Solady, the extension doesn't check for the code size of the contract, so that check should be added as well.

[L-03] Recommended use of ERC721 safeTransferFrom

The `withdrawERC721()` function in `HoneyLocker` uses `.transferFrom(address, address, uint256)`:

```
function withdrawERC721
  (address _token, uint256 _id) external onlyUnblockedTokens(_token) onlyOwner {
    ERC721(_token).transferFrom(address(this), msg.sender, _id);
}
```

This might lead to two issues:

- The caller might not be able to react to an expected NFT transfer, as no callback is performed
- Some ERC20 tokens do not decrease the allowance if the `msg.sender` is the one owner of the tokens. In those cases, and because `transferFrom(address, address, uint256)` shares the same signatures for ERC20s and ERC721s, it would be possible to transfer those tokens out of the lock without any other restrictions.

Recommendation

Use the ERC721 `safeTransferFrom(address, address, uint256)` function. This way, a callback is performed, and no ERC20s can be transferred via this path.

[L-04] Empty calls can be made to any of the allowed contracts

It is possible to make calls with no data to any of the allowed contracts, bypassing the `onlyAllowedSelector` modifier.

The modifier checks the expected selector via `mload(add(_data, 32))`, but it never checks that the `_data` length is not zero.

```
modifier onlyAllowedSelector
  (address _stakingContract, string memory action, bytes memory _data) {
    bytes4 selector;
    assembly {
        selector := mload(add(_data, 32))
    }
    if (!HONEY_QUEEN.isSelectorAllowedForTarget
      (selector, action, _stakingContract)) {
        revert SelectorNotAllowed();
    }
    _;
}
```

So it is possible to pass an empty `_data = ""`, and append the expected selector to the end of the `calldata` passed to the function.

For example for the `wildcard()` function we could call it like this:

```
bytes4 finalizeRedeemSelector = bytes4(keccak256("finalizeRedeem(uint256)"));

bytes memory extendedData = abi.encodePacked(
    abi.encodeWithSignature("wildcard(address,bytes)", address(xKDK), ""),
    abi.encode(finalizeRedeemSelector)
)

(bool ok,) = address(honeyLocker).call(extendedData);
require(ok);
```

So `selector := mload(add(_data, 32))` will read the whitelisted selector, but the function will make an empty `.call("")`.

```
function wildcard(address _contract, bytes calldata _data) onlyAllowedSelector
  (_contract, "wildcard", _data) ... {
    (bool success,) = _contract.call(_data); // @audit `.call("")`
    if (!success) revert WildcardFailed();
}
```

If the target contract has a `fallback()` or a non-payable `receive()`, the execution will succeed (also if the target were an EOA, although unlikely).

This can happen in any of the `stake()`, `unstake()`, `wildcard()`, and `claimRewards()` functions. The `stake()` function would be the most problematic, as it approves tokens that may not be used by the target contract. It also increases the `staked` storage variable and emits a misleading event.

Recommendation

Verify that `_data.length >= 4`

[L-05] HoneyQueen emits wrong old values in events

When setting new values for the `HoneyQueen` contract, a wrong one is emitted for the old attribute.

For example:

```
function setTreasury(address _treasury) external onlyOwner {
    treasury = _treasury;
    emit TreasurySet(treasury, _treasury);
}
```

`treasury` is assigned `_treasury` before emitting the event. So both variables will hold the same value on `TreasurySet`. The old value is lost basically.

This happens for the following functions:

- `setTreasury()`
- `setFees()`
- `setValidator()`
- `setAutomaton()`

Recommendation

Log the old value accordingly. Here's a suggested implementation:

```
function setTreasury(address _treasury) external onlyOwner {
    address oldTreasury = _treasury;
    treasury = _treasury;
    emit TreasurySet(oldTreasury, _treasury);
}
```

[L-06] Calldata arguments are not checked on `stake`/`unstake`

When the owner of `HoneyLocker` wants to stake or unstake, an argument called `_data` is passed to execute the transaction that will complete the action of staking or unstaking.

From that `_data` argument, only the selector is checked in the modifier `onlyAllowedSelector`, but the rest of the data is not checked at all. This can allow an owner to bypass the `expirations` mapping by staking/unstaking in a protocol and setting a different address as the receiver.

Currently, the protocols integrated with Interpol don't have this feature of setting a different receiver than the `msg.sender`, but it's relatively common to see this pattern in defi. For example, in Aave it is possible to supply and withdraw funds to another address different than `msg.sender`:

```
function supply(
    address asset,
    uint256 amount,
>>  address onBehalfOf,
    uint16 referralCode
  ) public virtual override {
// ...
  function withdraw(
    address asset,
    uint256 amount,
>>  address to
  ) public virtual override returns (uint256) {
```

If in the future Interpol integrates protocols that offer this feature, this issue will become exploitable and will allow the owner of a `HoneyLocker` to completely bypass the lock in LP tokens (`expirations` mapping).

[L-07] Approvals should be revoked when unstaking in HoneyLocker.sol

In `HoneyLocker.stake()`, the function will approve the staking contract with an arbitrary amount.

```
function stake(
  address_LPToken,
  address_stakingContract,
  uint256_amount,
  bytesmemory_data
)
        external
        onlyOwner
        onlyAllowedTargetContract(_stakingContract)
        onlyAllowedSelector(_stakingContract, "stake", _data)
    {
        staked[_LPToken][_stakingContract] += _amount;
 >      ERC20(_LPToken).approve(address(_stakingContract), _amount);
```

When unstaking, the approval is not nullified.

```
function unstake(
    address_LPToken,
    address_stakingContract,
    uint256_amount,
    bytesmemory_data
  )
        public
        onlyOwner
        onlyAllowedTargetContract(_stakingContract)
        onlyAllowedSelector(_stakingContract, "unstake", _data)
    {
        staked[_LPToken][_stakingContract] -= _amount;
        (bool success,) = _stakingContract.call(_data);
        if (!success) revert UnstakeFailed();
```

Set approvals to zero to prevent the staking contract from having lingering approvals from the HoneyLocker contracts. Also, it will be good in the staking function to approve to zero first before approving the amount for best practice, in case the token requires approval to zero to work.

```
ERC20(_LPToken).approve(address(_stakingContract), 0);
```

[L-08] Returning the fee share for an account that is not a referrer

In Beekeeper.sol, the getter function `referrerFeeShare()` will return `standardReferrerFeeShare` for an address that is not a referrer. Instead, it should return 0.

```
/// @notice Returns the fee share for a given referrer
    /// @dev If a custom fee share is set for the referrer, it returns that
    // value.
    ///      Otherwise, it returns the standard referrer fee share.
    /// @param _referrer The address of the referrer
    /// @return The fee share for the referrer in basis points (bps)
    function referrerFeeShare(address _referrer) public view returns (uint256) {
>
        return _referrerFeeShare[_referrer] != 0 ? _referrerFeeShare[_referrer] : sta
    }
```

[L-09] The amount of tokens staked per contract can be artificially inflated

When calling `stake()` in HoneyLocker.sol, the function takes in an `_amount` and adds it to the `staked` mapping. The function then calls the `_stakingContract` with the `_data` parameter.

```
function stake(
  address_LPToken,
  address_stakingContract,
  uint256_amount,
  bytesmemory_data
)
        external
        onlyOwner
        onlyAllowedTargetContract(_stakingContract)
        onlyAllowedSelector(_stakingContract, "stake", _data)
    {
>       staked[_LPToken][_stakingContract] += _amount;
        ERC20(_LPToken).approve(address(_stakingContract), _amount);
>       (bool success,) = _stakingContract.call(_data);
        if (!success) revert StakeFailed();

        emit Staked(_stakingContract, _LPToken, _amount);
    }
```

`_amount` can be different from the actual amount that is being transferred to the staking contract. The owner can set `_amount` as an arbitrarily large amount while the actual LP tokens sent to the staking contract is 1e18.

In the test contract, the owner can set `_amount` as 10000e50 and stake only 1e18 `HONEYBERA_LP`.

```
honeyLocker.stake(
            address(HONEYBERA_LP),
            address(KODIAK_STAKING),
>           10000e50,
>           abi.encodeWithSelector(bytes4(keccak256("stakeLocked
  (uint256,uint256)")), 1e18, 30 days)
        );
```

The `staked[_LPToken][_stakingContract]` will be 10000e50 while the actual stake amount is 1e18.

Similarly in `unstake()`, the `_amount` parameter can be set to zero, and the actual unstake amount is set at 1e18 (or in Kodiak's case, `withdrawLockedAll()` withdraws all the LP tokens in the Kodiak staking contract).

If the `staked` mapping is needed, consider checking the amount staked using the `address(honeyLocker)` balance instead. Check the balance before and after the call, and update the `staked` mapping accordingly.

[L-10] Expiration timing can be set to before the current time

When depositing LP tokens into HoneyLocker.sol, the user will call `depositAndLock()`, and set the `_expiration` time. This `_expiration` parameter is not checked, so the caller can set a time before `block.timestamp`.

```
> function depositAndLock(
  address_LPToken,
  uint256_amountOrId,
  uint256_expiration
) external onlyOwnerOrMigratingVault {
        // we only allow subsequent deposits of the same token IF the
        // expiration is the same or greater
>       if
  (!unlocked && expirations[_LPToken] != 0 && _expiration < expirations[_LPToken]) {
            revert ExpirationNotMatching();
        }
        // set expiration to 1 so token is marked as lp token
        expirations[_LPToken] = unlocked ? 1 : _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);
```

Tokens can then be withdrawn if `block.timestamp > expiration`.

```
function withdrawLPToken
   (address _LPToken, uint256 _amount) external onlyOwner {
        if (expirations[_LPToken] == 0) revert HasToBeLPToken();
        // only withdraw if expiration is OK
>       if (block.timestamp < expirations[_LPToken]) revert NotExpiredYet();
        ERC20(_LPToken).transfer(msg.sender, _amount);
        emit Withdrawn(_LPToken, _amount);
    }
```

Recommend adding a check in `depositAndLock()`.

```
require(_expiration > block.timestamp, "Wrong time");
```