

Migration to new HoneyLocker may revert if previous referral has changed in new HoneyLocker due to malicious activity

Description

In InterPoL protocol referrers are registered actors who receives 25% of all fees collected over the lifetime of a locker. Referrers are set only once during locker creation and there is no other way to update the referrer's address.

If we see the migrate() in HoneyLocker contract the migration will revert if referrer of new HoneyLocker contract does not match with the current referrer of current HoneyLocker contract.

```
if (
    HoneyLocker(_newHoneyLocker).unlocked() != unlocked ||
    address(HoneyLocker(_newHoneyLocker).HONEY_QUEEN()) !=
address(HONEY_QUEEN) ||
@>        HoneyLocker(_newHoneyLocker).referral() != referral
) {
    revert WrongTargetVaultParameters();
}
```

But a referrer can be malicious, if so then keeping the same referrer in new HoneyLocker contract will be very risky for the protocol. But as there is no functionality to change the referrer it is not possible to deploy the new HoneyLocker with another referrer, because if it is done then migrate() will revert due to above mentioned check.

Recommendation

It is better to remove the referral check from above code.

```
if (
    HoneyLocker(_newHoneyLocker).unlocked() != unlocked ||
    address(HoneyLocker(_newHoneyLocker).HONEY_QUEEN()) !=
address(HONEY_QUEEN) ||
-        HoneyLocker(_newHoneyLocker).referral() != referral
) {
    revert WrongTargetVaultParameters();
}
```

The HoneyLocker.sol contract is not compatible with Infrared ecosystem for which iBGT will remain

partially unused

Summary

The HoneyLocker.sol contract is designed to deal with BGT token because staking protocols like BGT Station returns reward in BGT form which the owner/operator can delegate to validator to earn rewards. But the case is little different for staking protocol like Infrared where rewards are paid using iBGT which is the liquid wrapper of BGT. As the HoneyLocker.sol contract is not configured to work with iBGT the iBGT will remain unused.

Finding Description

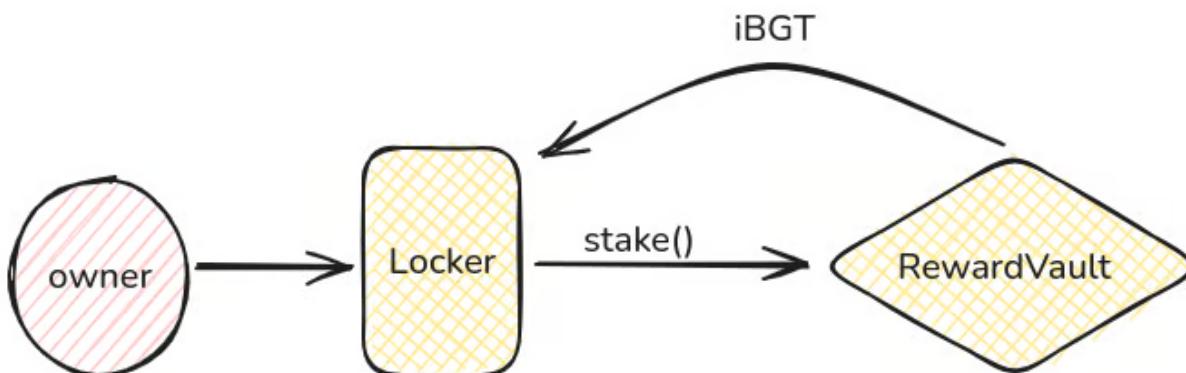
Infrared staking protocol is one of the protocols with whom the InterPoL interacts.

We know that the HoneyLocker contract works with LP tokens and related reward vaults, each LP tokens has a separate reward vaults, for ex- HONEY-WBERA-LP will have HONEY-WBERA-vault. Users i.e owner/operator will use the HoneyLocker contract like this:

Owner will deposit the LP token in locker using `depositAndLock()`. Then they will call `stake()` to deposit the LP token in reward vault. For that the vault issues reward in the form of BGT to the staker i.e in our case the locker. The owner/operator can use the BGT to delegate validators by which they earn some type of rewards. This is the general process of using the locker [for BGT station the locker works like this].

But there is little difference for the Infrared staking protocol. The process of using this protocol is like :

Users deposit their liquidity in BERA/HONEY LP vault. For that they earn LP yield & receives iBGT. Users can use this iBGT in Defi. Users can also stake the iBGT to earn the delegation reward associated with BGT.



i.e the reward vault will return the iBGT as reward. This iBGT is tradeable so users can use this on trade or stake it to earn delegation reward. So here we have two options with iBGT - either 1. we can use it in defi or 2. we can use it to delegate by staking it in iBGT staking pool.

Lets go with the 1st option: After receiving the iBGT as reward the only function[in HoneyLocker.sol] allow us to withdraw that iBGT is `withdrawERC20()`, but this contract does a approval operation. We tried but could find the contract for iBGT. If the iBGT is normal ERC20 contract then it will work fine, but if the iBGT

requires governance interference in approval, just like BGT, then it will not work i.e the tx will fail because governance approves selected address to allow calling approve.

For 2nd option: To stake the iBGT in iBGT staking pool the locker must have related function which it does not have currently. One thing is possible if the owner/operator manages to withdraw the iBGT token from withdrawERC20() then they can stake it. But there may be an issue with this, see this from here:

Users can also choose stake their iBGT to earn the yield associated with the underlying BGT, including vote incentives and rewards from native dapps (e.g Berps).

Here owner/operator can earn yield associated with the underlying BGT. When LP was staked in the vault the locker contract was the staker, and locker is considered the owner of BGT and but as owner/operator is not the staker for the reward vault they can't stake [possibly] & earn validator reward.

If so then the iBGT could not be used properly.

Impact Explanation The impact is high because an important feature of iBGT will not be possible to use.

Likelihood Explanation Likelihood is high because there will be deposits using Infrared staking protocol.

Proof of Concept

<https://infrared.finance/blog/understanding-ibgt-and-its-rewards> BGT Management section in HoneyLocker.sol contract.

Recommendation

Implement some feature to stake iBGT from this contract.

The migrate() call will fail because depositAndLock() on new HoneyLocker contract can only be called by the creator of locker

Summary

Migration will fail because of depositAndLock() call on new HoneyLocker contract inside the migration().

Finding Description

It is visible that depositAndLock() only be called by either owner or operator or migrating vault. So if the new HoneyLocker contract's owner is set with any other address except old HoneyLocker then the migrate() call will revert. Because when, from inside the migrate(), the depositAndLock() is called on new HoneyLocker contract, as the depositAndCall() expects the call from owner, in our case any address which was set as owner [except the old HoneyLocker], the call will revert because here the caller is old HoneyLocker contract.

Run this test in HoneyLocker.t.sol file:

```

function test_myMigration() public {
    // deposit some LP token in honeyLocker
    uint256 balance = HONEYBERA_LP.balanceOf(THJ);
    vm.prank(THJ);
    HONEYBERA_LP.approve(address(honeyLocker), balance);
    vm.prank(THJ);
    honeyLocker.depositAndLock(address(HONEYBERA_LP), balance,
expiration);

    // deploy new honeyLocker from factory
    HoneyLockerV2 honeyLockerV2 =
HoneyLockerV2(payable(factory.clone(THJ, referral))); // @audit THJ is set
as the owner

    // set migration flag
    vm.prank(THJ);
    honeyQueen.setMigrationFlag(true, address(honeyLocker).codehash,
address(honeyLockerV2).codehash);

    // migrate
    vm.prank(THJ);
    vm.expectRevert();
    honeyLocker.migrate(SLA.addresses(address(HONEYBERA_LP)),
SLA.uint256s(balance), payable(address(honeyLockerV2)));
}

```

If we see in the documentation about how to create a locker it is understandable that the caller/user will be the owner of the locker. If so then the call to migrate will revert.

But if the old HoneyLocker is set as the owner of new locker [which is not possible] then the migrate() call will pass, but for that the old HoneyLocker contract must transfer the ownership to an EOA or any address. For this check run this test in HoneyLocker.t.sol file:

```

function test_myAnotherMigration() public {
    // deposit some LP token in honeyLocker
    uint256 balance = HONEYBERA_LP.balanceOf(THJ);
    vm.prank(THJ);
    HONEYBERA_LP.approve(address(honeyLocker), balance);
    vm.prank(THJ);
    honeyLocker.depositAndLock(address(HONEYBERA_LP), balance,
expiration);

    // deploy new honeyLocker from factory
    HoneyLockerV2 honeyLockerV2 =
HoneyLockerV2(payable(factory.clone(address(honeyLocker), referral))); // @audit see that old HoneyLocker was set as owner

    // set migration flag
    vm.prank(THJ);
    honeyQueen.setMigrationFlag(true, address(honeyLocker).codehash,

```

```
address(honeyLockerV2).codehash);

    // migrate
    vm.prank(THJ);
    honeyLocker.migrate(SLA.addresses(address(HONEYBERA_LP)),
SLA.uint256s(balance), payable(address(honeyLockerV2)));

    vm.prank(address(honeyLocker));
    honeyLockerV2.transferOwnership(THJ);
    assertEq(honeyLockerV2.owner(), THJ);
}
```

To make this possible the HoneyLocker contract must have a function to transfer the ownership from itself to someone which it does not have.

Impact Explanation

The impact is high because migration will never be possible. We assume old HoneyLocker contract cannot be set as owner when new HoneyLocker will be created.

Likelihood Explanation

Likelihood is high because all migrate() call will fail.

Proof of Concept

Was given in description.