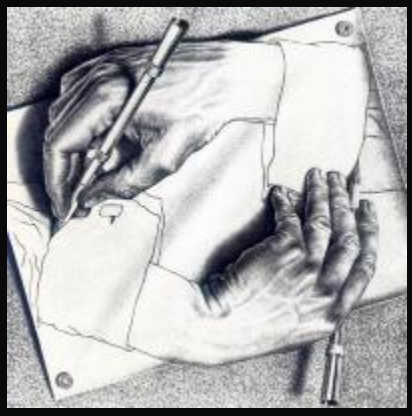
certora

Security Assessment Report



Doppler

November 2024

Prepared for Whetstone Research

certora

Table of content

Project Summary **3**
- Project Scope 3
- Project Overview 3
  - Protocol Overview 4
- Findings Summary 4
- Severity Matrix 4

Detailed Findings **5**
- Critical Severity Issues 8
- C-01 Missing PoolKey validation allows complete fund drain from Airlock 8
- C-02 The Doppler contract can be reinitialized from the PoolManager 15
- C-03 In UniswapV2Migrator.migrate(): misconfiguration of recipient to msg.sender in addLiquidityETH call 19
- C-04 In UniswapV2Migrator.migrate(): misuse of amount0 instead of amount1 in addLiquidityETH call 20
- High Severity Issues 21
- H-01 An arbitrary Airlock contract can be set due to hookData not being validated and matched with poolKey 21
- H-02 Anyone can call beforeInitialize() directly 24
- H-03 Donations are possible by default and should be prevented here 26
- H-04 Missing receive function in UniswapV2Migrator will lead to addLiquidityETH reverting 28
- H-05 Lack of access control and recipient validation in UniswapV2Migrator.migrate() can drain funds held by the contract 30
- H-06 UniswapV2Migrator.migrate() is griefable by frontrunning pool creations on Uniswap V2 32
- H-07 In UniswapV2Migrator.migrate(), the call to router.addLiquidity() actually inputs 0 for the amounts 35
- H-08 Token creation is frontrunnable and can grief users' calls to Airlock.create() 36
- H-09 Hook creation is frontrunnable and can grief users' call to Airlock.create() 38
- H-10 Airlock.create() is itself frontrunnable and can grief users' call to Airlock.create() 40
- H-11 approve() may revert if the current approval is not zero 41
- Medium Severity Issues 43
- M-01 Lack of slippage protection in UniswapV2Migrator.migrate() risks value loss 43
- M-02 deadline == block.timestamp is ineffective 44
- M-03 Return value of transfer() not checked 45
- M-04 Incorrect loop start index in for positions in _unlockCallback() 47
- Low Severity Issues 48
- L-01 Some tokens may revert when zero value transfers are made 48
- **Informational Severity Issues** **50**
- I-01. The slugs array size returned by _computePriceDiscoverySlugsData() is too big 50

Disclaimer **52**

About Certora **52**

certora

Project Summary

Project Scope

Project Name	Repository (link)	Latest Commit Hash	Platform
Doppler	https://github.com/whetstoneresearch/doppler-smart-contracts/	047e6297c0a04541856b45ed2d50849b22c015ad	EVM

Project Overview

This document describes the verification of **Doppler** using manual code review findings. The work was undertaken from **November 4** to **November 11, 2024**

The following contract list is included in our scope:

- Doppler.sol
- Airlock.sol
- Governance.sol
- UniswapV2Migrator.sol
- DopplerFactory.sol
- DERC20.sol
- GovernanceFactory.sol
- TokenFactory.sol
- IHookFactory.sol
- ITokenFactory.sol
- IMigrator.sol
- IGovernanceFactory.sol

The team performed a manual audit of all the Solidity contracts**.** During the manual audit, the Certora team discovered bugs in the code, as listed on the following page.

certora

Protocol Overview

Doppler is a liquidity bootstrapping protocol built on top of Uniswap v4. Doppler abstracts the challenges associated with Uniswap v4 integrations from integrators by executing the entire liquidity bootstrapping auction inside of the hook contract.

Findings Summary

The table below summarizes the findings of the review, including type and severity details.

Severity	Discovered	Confirmed	Fixed
Critical	4	4	4
High	11	11	11
Medium	4	4	4
Low	1	1	1
Informational	1	1	1
Total	21	21	21

Severity Matrix

Impact	High	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Low	Low	Medium
		Low	Medium	High
		Likelihood		

certora

Detailed Findings

ID	Title	Severity	Status
C-01	Missing PoolKey validation allows complete fund drain from Airlock	Critical	Fixed
C-02	The Doppler contract can be reinitialized from the PoolManager	Critical	Fixed
C-03	In UniswapV2Migrator.migrate(): misconfiguration of recipient to msg.sender in addLiquidityETH call	Critical	Fixed
C-04	In UniswapV2Migrator.migrate(): misuse of amount0 instead of amount1 in addLiquidityETH call	Critical	Fixed
H-01	An arbitrary Airlock contract can be set due to hookData not being validated and matched with poolKey	High	Fixed
H-02	Anyone can call beforeInitialize() directly	High	Fixed
H-03	Donations are possible by default and should be prevented here	High	Fixed
H-04	Missing receive function in UniswapV2Migrator will lead to	High	Fixed

	addLiquidityETH reverting		
H-05	Lack of access control and recipient validation in UniswapV2Migrator.migrate() can drain funds held by the contract	High	Fixed
H-06	UniswapV2Migrator.migrate() is griefable by frontrunning pool creations on Uniswap V2	High	Fixed
H-07	In UniswapV2Migrator.migrate(), the call to router.addLiquidity() actually inputs 0 for the amounts	High	Fixed
H-08	Token creation is frontrunnable and can grief users' calls to Airlock.create()	High	Fixed
H-09	Hook creation is frontrunnable and can grief users' call to Airlock.create()	High	Fixed
H-10	Airlock.create() is itself frontrunnable and can grief users' call to Airlock.create()	High	Fixed
H-11	approve() may revert if the current approval is not zero	High	Fixed
M-01	Lack of slippage protection in UniswapV2Migrator.migrate() risks value loss	Medium	Fixed
M-02	deadline == block.timestamp is	Medium	Fixed

	ineffective		
M-03	Return value of transfer() not checked	Medium	Fixed
M-04	Incorrect loop start index in for positions in _unlockCallback()	Medium	Fixed
L-01	Some tokens may revert when zero value transfers are made	Low	Fixed

Critical Severity Issues

C-01 Missing PoolKey validation allows complete fund drain from Airlock

Severity: **Critical**	Impact: **High**	Likelihood: **High**
Files: Airlock.sol UniswapV2Migrator.sol	Status: Fixed	Violated Property: N/A

Description

The `Airlock` contract is used to deploy new tokens with the associated governance, timelock and hook contracts. Then, when relevant, it triggers the migration from the Doppler hook to another liquidity pool.

The issue here stems from the fact that in `Airlock.create()`, the `PoolKey memory poolKey` input is never validated. Therefore, arbitrary values are possible, including an arbitrary hooks contract. Afterwards, `Airlock.migrate()` can be called in the same transaction using arbitrary values returned by the malicious hooks contract and drain all funds from the `Airlock` contract.

Exploit Scenario:

1. Our attacker deploys a malicious hook contract
2. Our attacker can call `Airlock.create()` with the address `poolKey.hooks` pointing to their malicious contract. Notice that no input validation exists on the poolkey:

```
File: Airlock.sol
065:      * @param poolKey Pool key of the liquidity pool (precomputed)
...
077:     function create(
...
082:         PoolKey memory poolKey,
```

```
...
116:         getTokenData[token] = TokenData({
117:             governance: governance,
118:             recipients: recipients,
119:             amounts: amounts,
120:             migrator: migrator,
121:             timelock: timelock,
122:             poolKey: poolKey
123:         });
...
126:         poolManager.initialize(poolKey, TickMath.getSqrtPriceAtTick(0),
new bytes(0));
127:
128:         emit Create(token, poolKey, hook);
```

The `PoolManager` contract from Uniswap V4 will create a new PoolId, but our Doppler contract is not getting initialized (hooks aren't pointing to the deployed Doppler contract; it's only assumed to have been precomputed).

Additionally, in the attacker's call to `Airlock.create()`, they can input themselves as the sole recipient for the whole amount of their newly deployed token minted. This is important as it will enable them to take control of the timelock contract to which all of `Airlock`'s funds will be sent during the fake migration.

```
File: Airlock.sol
66:      * @param recipients Array of addresses to receive tokens after the
migration
67:      * @param amounts Array of amounts to receive after the migration
...
77:     function create(
...
83:         address[] memory recipients,
84:         uint256[] memory amounts,
...
File: Airlock.sol
107:         address token = tokenFactory.create(name, symbol, initialSupply,
address(this), address(this), tokenData, salt);
108:         address hook = hookFactory.create(poolManager, numTokensToSell,
hookData, salt);
```

```
109:
110:         ERC20(token).transfer(hook, numTokensToSell);
...
113:         (address governance, address timelock) =
governanceFactory.create(name, token, governanceData);
114:         Ownable(token).transferOwnership(timelock);
115:
116:         getTokenData[token] = TokenData({
117:             governance: governance,
118:             recipients: recipients,
119:             amounts: amounts,
120:             migrator: migrator,
121:             timelock: timelock,
122:             poolKey: poolKey
123:         });
```

It's important to note that the newly deployed Doppler hook is supposed to contain an amount of `numTokensToSell` tokens that can't be retrieved and that we don't expect to ever retrieve. This is why the value of `numTokensToSell` could be 0 here or anything arbitrary. The attacker will get the token that will give them control of the governance through the use of `initialSupply` being equal to the `amount` they'll receive directly on migration (or at least enough to pass all propositions)

The attacker can now call `Airlock.migrate()` (possibly in the same transaction), and here's a breakdown of what will happen:

Recovering all of the newly created tokens to gain control of the timelock:

```
File: Airlock.sol
133:     /**
134:      * @notice Triggers the migration from the Doppler hook to another
liquidity pool
135:      * @param asset Address of the token to migrate
136:      */
137:     function migrate(
138:         address asset
139:     ) external {
140:         TokenData memory tokenData = getTokenData[asset];
141:
```

```
142:         uint256 length = tokenData.recipients.length;
143:         for (uint256 i; i < length; i++) {
144:             ERC20(asset).transfer(tokenData.recipients[i],
tokenData.amounts[i]);  //@audit 1 recipient being the attacker and 1 amount
being initialSupply
145:         }
```

Returning arbitrary values from their malicious hook contract

```
File: Airlock.sol
147:         (uint256 amount0, uint256 amount1) =
IHook(address(tokenData.poolKey.hooks)).migrate();  //@audit amount 0 and
amount 1 can be anything
```

Using arbitrary currencies

These weren't input-validated either so anything can be returned. Notice that a Uniswap V4 Pool exists with these currencies and the malicious hooks contract used to compute a PoolId but funds were never sent.

```
File: Airlock.sol
149:         address currency0 = Currency.unwrap(tokenData.poolKey.currency0);
150:         address currency1 = Currency.unwrap(tokenData.poolKey.currency1);
```

Sending all funds to the migrator contract which points to their timelock contract of the attackeras the recipient

Some examples of possible inputs:

- `currency0` could be USDC, `amount0` could be the Airlock contract's USDC balance (belonging to all other users)
- `currency0` could be address(0), `amount0` could be the Airlock contract's Ether balance (belonging to all other users).

- `currency1` could be WBTC, `amount1` could be the Airlock contract's WBTC balance (belonging to all other users)

```
File: Airlock.sol
152:         if (currency0 != address(0))
ERC20(currency0).transfer(address(tokenData.migrator), amount0);
153:         ERC20(currency1).transfer(address(tokenData.migrator), amount1);
154:
155:         (address pool,) = tokenData.migrator.migrate{ value: currency0 ==
address(0) ? amount0 : 0 }(
156:             currency0, currency1, amount0, amount1, tokenData.timelock,
new bytes(0)
157:         );
```

Notice how `Airlock.migrate()` isn't `payable` so the funds are received through the `receive()` function, making it possible for an attacker to lurk in wait for a big transfer of funds and backrun it by calling `migrate()`.
Notice also that the attack is quite repeatable for the same currencies: the attacker only needs to transfer their DERC20 asset to the `Airlock` contract to pass the `ERC20(asset).transfer(tokenData.recipients[i], tokenData.amounts[i]);` call and then chose again any arbitrary value to return through the call to `migrate()` on their malicious hook. They only need to deploy as pools pointing to their malicious hook contract as there are different currencies held in the Airlock contract (also doable in 1 transaction), and this will give them enough power to attack multiple times.

UniswapV2Migrator.migrate() will transfer all of the funds to the timelock contract of the attacker

```
File: UniswapV2Migrator.sol
61:     function migrate(
62:         address token0,
63:         address token1,
64:         uint256 amount0,
65:         uint256 amount1,
66:         address recipient,
67:         bytes memory
```

```
68:     ) external payable returns (address pool, uint256 liquidity) {
69:         pool = factory.getPair(token0, token1);  //@audit it doesn't
matter if the pair exists or not
70:
71:         if (pool == address(0)) {
72:             pool =
73:                 factory.createPair(token0 == address(0) ?
0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2 : token0, token1);
74:         }
75:
76:         if (token0 != address(0)) ERC20(token0).approve(address(router),
amount0); //@audit amount0 here would be the whole balance of currency0
previously held by the Airlock contract
77:         ERC20(token1).approve(address(router), amount1); //@audit amount1
here would be the whole balance of currency1 previously held by the Airlock
contract
78:
79:         if (token0 == address(0)) {
80:             (,, liquidity) = router.addLiquidityETH{ value: amount0
}(token1, amount0, 0, 0, recipient, block.timestamp);  //@audit this will
transfer Ether (in the case of currency0 being address(0)) and mint LP tokens
to the timelock contract controlled by the attacker
81:             SafeTransferLib.safeTransferETH(recipient,
address(this).balance); //@audit this will transfer any leftover directly to
the timelock contract controlled by the attacker
82:         } else {
83:             ERC20(token0).approve(address(router), amount0); //@audit
amount0 here would be the whole balance of currency0 (token0) previously held
by the Airlock contract
84:             (,, liquidity) = router.addLiquidity(token0, token1, 0, 0, 0,
0, msg.sender, block.timestamp);  //@audit recipient here is mistakenly the
Airlock contract but no liquidity is being added
85:             SafeTransferLib.safeTransfer(ERC20(token0), recipient,
ERC20(token0).balanceOf(address(this))); //@audit all of the token0 balance
previously held by the Airlock contract will be sent to the timelock contract
controlled by the attacker
86:         }
87:
88:         SafeTransferLib.safeTransfer(ERC20(token1), recipient,
ERC20(token1).balanceOf(address(this)));//@audit all of the remaining token1
balance will be sent to the timelock contract controlled by the attacker
89:     }
```

The attacker can now profit

They're the only owner of governance tokens for the timelock contract and have full voting power to recover all funds

Recommendations:

- Validate that the `token` address generated from `address token = tokenFactory.create(name, symbol, initialSupply, address(this), address(this), tokenData, salt);` is the same as either `poolKey.currency0` or `poolKey.currency1` depending on the currencies' alphabetical order.
- Validate that the `hook` address generated from `address hook = hookFactory.create(poolManager, numTokensToSell, hookData, salt);` is the same as `poolKey.hooks`

Whetstone's response:

Fixed

C-02 The Doppler contract can be reinitialized from the PoolManager

Severity: **Critical**	Impact: **High**	Likelihood: **High**
Files: Doppler.sol	Status: Fixed	Violated Property: N/A

Description

The `Doppler` contract should validate that the calls are coming regarding the right pool. Indeed, especially here where `isInitialized` isn't set, it'd be possible to create a new pool on Uniswap V4's PoolManager which would point to an existing Doppler contract.

Let's assume that the `beforeInitialize` mitigates the bug regarding the missing `onlyPoolManager` modifier but not the lack of setting `isInitialized` to true:

```
File: Doppler.sol
180:     function beforeInitialize(
181:         address,
182:         PoolKey calldata key,
183:         uint160,
184:         bytes calldata
- 185:     ) external override returns (bytes4) {
+ 185:     ) external override onlyPoolManager returns (bytes4) {
```

It would be possible to override the existing `poolKey` by initializing a new pool on Uniswap V4's PoolManager by setting the `hooks` field to the existing Doppler contract:

```
File: Doppler.sol
180:     function beforeInitialize(
181:         address,
182:         PoolKey calldata key,
```

```
183:         uint160,
184:         bytes calldata
185:     ) external override returns (bytes4) { //@audit-issue Lacking
onlyPoolManager modifier
186:         if (isInitialized) revert AlreadyInitialized();  //@audit-issue
never set
187:         poolKey = key; //@audit-issue can be overriden even if the
onlyPoolManager modifier was there
188:
189:         // Enforce maximum tick spacing
190:         if (key.tickSpacing > MAX_TICK_SPACING) revert
InvalidTickSpacing();
191:
192:         /* Gamma checks */
193:         // Enforce that the total tick delta is divisible by the total
number of epochs
194:         // Enforce that gamma is divisible by tick spacing
195:         if (gamma % key.tickSpacing != 0) revert InvalidGamma();
196:
197:         return BaseHook.beforeInitialize.selector;
198:     }
```

Then `afterInitialize` will trigger the call to `_unlockCallback`:

```
File: Doppler.sol
204:     function afterInitialize(
205:         address sender,
206:         PoolKey calldata key,
207:         uint160,
208:         int24 tick,
209:         bytes calldata
210:     ) external override onlyPoolManager returns (bytes4) {
211:         poolManager.unlock(abi.encode(CallbackData({ key: key, sender:
sender, tick: tick, isMigration: false })));
212:         return BaseHook.afterInitialize.selector;
213:     }
```

Which will enable the attacker to either influence the prices before migration (through passed `tick`) and reset `state.lastEpoch` to 1:

```
File: Doppler.sol
1002:             state.lastEpoch = 1; //@audit-issue this is reset here,
while this value is used in _rebalance to compute the accumulatorDelta
1003:
1004:             (, int24 tickUpper) = _getTicksBasedOnState(0,
key.tickSpacing);  //@audit-issue tickSpacing was only validated through
enforcing that gamma is divisible by tick spacing
1005:             uint160 sqrtPriceNext = TickMath.getSqrtPriceAtTick(tick);
//@audit-issue this is using a malicious tick
1006:             uint160 sqrtPriceCurrent =
TickMath.getSqrtPriceAtTick(tick);  //@audit-issue this is using a malicious
tick
1007:
1008:
1009:             SlugData memory lowerSlug = SlugData({ tickLower: tick,
tickUpper: tick, liquidity: 0 });  //@audit-issue manipulated slug
1010:             (SlugData memory upperSlug, uint256 assetRemaining) =
_computeUpperSlugData(key, 0, tick, numTokensToSell); //@audit-issue
manipulated slug
1011:             SlugData[] memory priceDiscoverySlugs =
1012:                 _computePriceDiscoverySlugsData(key, upperSlug,
tickUpper, assetRemaining); //@audit-issue manipulated slugs
1013:
```

Notice that `isMigration` is hardcoded to false, so the only opened path is this one.

Recommendations:

- Given that `PoolKey` is known before the deployment, its PoolId, as computed on Uniswap V4, could be one of the immutable values set in the constructor. This would make sure that only the relevant Pool can call this contract.
- Set `isInitialized` to true after initialization
- Add `onlyPoolManager` on `beforeInitialize`

Whetstone's response:

Valid finding, fixes were made in the following PRs:

- https://github.com/whetstoneresearch/doppler-smart-contracts/pull/205
- https://github.com/whetstoneresearch/doppler-smart-contracts/pull/206
- https://github.com/whetstoneresearch/doppler-smart-contracts/pull/53

Fix Review:

Should be good enough. The first recommendation isn't applied and could've been an additional protection but there's currently no way to set the `poolKey` elsewhere than in `beforeInitialize`, which is now protected.

C-03 In UniswapV2Migrator.migrate(): misconfiguration of recipient to msg.sender in addLiquidityETH call

Severity: **Critical**	Impact: **High**	Likelihood: **High**
Files: UniswapV2Migrator.sol	Status: Fixed	Violated Property: N/A

Description

`msg.sender` would be the airlock contract here instead of the token's timelock contract (`recipient`). This will lock the funds inside the Airlock contract instead of sending it to the right `recipient` (the token's governance's timelock)

Recommendations:

```
File: UniswapV2Migrator.sol
83:             ERC20(token0).approve(address(router), amount0);
- 84:             (,, liquidity) = router.addLiquidity(token0, token1, 0, 0,
0, 0, msg.sender, block.timestamp);
+ 84:             (,, liquidity) = router.addLiquidity(token0, token1, 0, 0,
0, 0, recipient, block.timestamp);
```

Whetstone's response:

Fixed

But this PR is more likely to be merged
https://github.com/whetstoneresearch/doppler-smart-contracts/pull/211

C-04 In UniswapV2Migrator.migrate(): misuse of amount0 instead of amount1 in addLiquidityETH call

Severity: **Critical**	Impact: **High**	Likelihood: **High**
Files: UniswapV2Migrator.sol	Status: Fixed	Violated Property: N/A

Description

- Funds are permanently locked if `amount0` > `amount1`
- Remaining funds are sent to the timelock contract if `amount0` < `amount1`

Recommendations:

```
File: UniswapV2Migrator.sol
- 80:             (,, liquidity) = router.addLiquidityETH{ value: amount0
}(token1, amount0, 0, 0, recipient, block.timestamp);
+ 80:             (,, liquidity) = router.addLiquidityETH{ value: amount0
}(token1, amount1, 0, 0, recipient, block.timestamp);
```

Whetstone's response:

Fixed, Valid finding. WIP PR here, the fix will likely be in the Airlock contract, not in the Uniswap migrator though:

High Severity Issues

H-01 An arbitrary Airlock contract can be set due to hookData not being validated and matched with poolKey

Severity: **High**	Impact: **High**	Likelihood: **Medium**
Files: Doppler.sol Airlock.sol DopplerFactory.sol	Status: Fixed	Violated Property: N/A

Note: The likelihood is medium due to the malicious Airlock being visible on chain. However, victims who don't know how to make checks on chain are valid targets.

Description

`hookData` is used in `Airlock.create()` to deploy a new Doppler contract through the DopplerFactory:

```
File: Airlock.sol
108:         address hook = hookFactory.create(poolManager, numTokensToSell,
hookData, salt);
```

```
File: DopplerFactory.sol
09: contract DopplerFactory is IHookFactory {
10:     function create(
11:         IPoolManager poolManager,
12:         uint256 numTokensToSell,
13:         bytes memory data,
14:         bytes32 salt
15:     ) external returns (address) {
16:         (
17:             uint256 minimumProceeds,
```

```
18:             uint256 maximumProceeds,
19:             uint256 startingTime,
20:             uint256 endingTime,
21:             int24 startingTick,
22:             int24 endingTick,
23:             uint256 epochLength,
24:             int24 gamma,
25:             bool isToken0,
26:             uint256 numPDSlugs,
27:             address migrator  //@audit-issue this should really be called
"airlock" instead as it could be misleading
28:         ) = abi.decode(data, (uint256, uint256, uint256, uint256, int24,
int24, uint256, int24, bool, uint256, address));
29:
30:         return address(
31:             new Doppler{ salt: salt }(
32:                 poolManager,
33:                 numTokensToSell,
34:                 minimumProceeds,
35:                 maximumProceeds,
36:                 startingTime,
37:                 endingTime,
38:                 startingTick,
39:                 endingTick,
40:                 epochLength,
41:                 gamma,
42:                 isToken0,
43:                 numPDSlugs,
44:                 migrator  //@audit-issue this should really be called
"airlock" instead as it could be misleading
45:             )
46:         );
```

However, it isn't validated and:

- `isToken0` can be set to the wrong boolean as there isn't a validation with `poolKey.currency0` and `poolKey.currency1`
- `migrator` is here expected to be the Airlock contract only but can be set to another value. This would make it so that a call `Doppler.migrate()` can be initiated through another `msg.sender` than the expected Airlock contract, which will therefore send funds directly to the arbitrary `msg.sender` instead of the Airlock contract:

```
File: Doppler.sol
1103:     function migrate() external returns (uint256 amount0, uint256
amount1) {
1104:         if (msg.sender != airlock) revert SenderNotAirlock();
//@audit-issue airlock can be anything
...
1112:             poolManager.unlock(
1113:                 abi.encode(CallbackData({ key: poolKey, sender:
msg.sender, tick: 0, isMigration: true }))
1114:             );
```

Whetstone's response:

- The name `migrator` was changed in this PR:
  https://github.com/whetstoneresearch/doppler-smart-contracts/pull/53
- Will add the other checks. Regarding the validation of the Airlock address, I will check how this can be done, since the address is passed around the HookFactory.
  https://github.com/whetstoneresearch/doppler-smart-contracts/pull/216

Fix Review:

On pull/216, only the check for `isToken0` is currently applied. To pass the actual Airlock address (and also mitigate frontrunning attacks), I suggest adding an access control mechanism so that only the Airlock contract can call. Then the field can be set to the expected Airlock.

H-02 Anyone can call beforeInitialize() directly

Severity: **High**	Impact: **High**	Likelihood: **Medium**
Files: Doppler.sol	Status: Fixed	Violated Property: N/A

Note: The likelihood is medium due to the existence of a workaround.

Description

`beforeInitialize` lacks the `onlyPoolManager` and `isInitialized` is never set

It means that any caller can pass some arbitrary values by directly calling `beforeInitialize` to change the `poolKey` without triggering other safety checks from the PoolManager:

```
File: Doppler.sol
179:
180:     function beforeInitialize(
181:         address,
182:         PoolKey calldata key,
183:         uint160,
184:         bytes calldata
185:     ) external override returns (bytes4) { //@audit-issue Lacking
onlyPoolManager modifier
186:         if (isInitialized) revert AlreadyInitialized();  //@audit-issue
never set
187:         poolKey = key;
...
197:         return BaseHook.beforeInitialize.selector;
```

Usually, the PoolKey is a calldata from the PoolManager but the one place it's used is in the `migrate()` function. This will break the migration flow and leave funds stuck:

```
File: Doppler.sol
1103:     function migrate() external returns (uint256 amount0, uint256
amount1) {
...
1112:             poolManager.unlock(
1113:                 abi.encode(CallbackData({ key: poolKey, sender:
msg.sender, tick: 0, isMigration: true }))  //@audit-issue arbitrary poolKey
1114:             );
```

Then `_unlockCallback` with `isMigration == true` is triggered and the PoolKey is arbitrary:

```
File: Doppler.sol
964:     function _unlockCallback(
965:         bytes calldata data
966:     ) internal override returns (bytes memory) {
967:         CallbackData memory callbackData = abi.decode(data,
(CallbackData));
968:         (PoolKey memory key, address sender, int24 tick, bool
isMigration) =
969:             (callbackData.key, callbackData.sender, callbackData.tick,
callbackData.isMigration);  //@audit-issue arbitrary key
970:
```

This can be used for grief migration.

Recommendations:

- A workaround exists where the rightful user would call `beforeInitialize()` themselves and `migrate()` in the same transaction so that the migration would work
- However `poolKey` can be prevented from changing if `onlyPoolManager` is applied and `isInitialized` is set

Whetstone's response:

Fixed

H-03 Donations are possible by default and should be prevented here

Severity: **High**	Impact: **Medium**	Likelihood: **High**
Files: Doppler.sol	Status: Fixed	Violated Property: N/A

Note: The impact is medium due to breaking a core invariant. The likelihood is high due to the ease of using the open path.

Description

The protocol doesn't want anyone to be able to provide liquidity as stated in the `beforeAddLiquidity` hook:

```
File: Doppler.sol
371:     /// @notice Called by the poolManager immediately before liquidity is
added
372:     ///         We revert if the caller is not this contract
373:     /// @param caller The address that called poolManager.modifyLiquidity
374:     function beforeAddLiquidity(
375:         address caller,
376:         PoolKey calldata,
377:         IPoolManager.ModifyLiquidityParams calldata,
378:         bytes calldata
379:     ) external view override onlyPoolManager returns (bytes4) {
380:         if (caller != address(this)) revert Unauthorized();
381:
382:         return BaseHook.beforeAddLiquidity.selector;
383:     }
```

However, the donation hooks aren't activated:

```
File: Doppler.sol
1083:     /// @notice Returns a struct of permissions to signal which hook
functions are to be implemented
1084:     function getHookPermissions() public pure override returns
(Hooks.Permissions memory) { //@audit-issue should prevent donation in
beforeDonate
1085:         return Hooks.Permissions({
1086:             beforeInitialize: true,
1087:             afterInitialize: true,
1088:             beforeAddLiquidity: true,
1089:             beforeRemoveLiquidity: false,
1090:             afterAddLiquidity: false,
1091:             afterRemoveLiquidity: false,
1092:             beforeSwap: true,
1093:             afterSwap: true,
1094:             beforeDonate: false, //@audit not activated
1095:             afterDonate: false, //@audit not activated
1096:             beforeSwapReturnDelta: false,
1097:             afterSwapReturnDelta: false,
1098:             afterAddLiquidityReturnDelta: false,
1099:             afterRemoveLiquidityReturnDelta: false
1100:         });
1101:     }
```

And, by default, donations are permitted on a Uniswap V4 pool.

Consider activating the `beforeDonate` hook to always revert on donations so as to avoid a potential price manipulation

Whetstone's response:

Fixed

H-04 Missing receive function in UniswapV2Migrator will lead to addLiquidityETH reverting

Severity: **High**	Impact: **High**	Likelihood: **Medium**
Files: UniswapV2Migrator.sol	Status: Fixed	Violated Property: N/A

Note: The likelihood is medium due to workarounds being possible with precise calculations.

Description

In `UniswapV2Migrator.migrate()`, there's a call to `addLiquidityETH`: followed by a direct transfer of leftover Eth (probably from returned dust):

```
File: UniswapV2Migrator.sol
80:             (,, liquidity) = router.addLiquidityETH{ value: amount0
}(token1, amount0, 0, 0, recipient, block.timestamp);
81:             SafeTransferLib.safeTransferETH(recipient,
address(this).balance);
```

The `addLiquidityETH` function on UniswapV2Router02.sol#L77-L100 refunds dust Eth if there are some:

```
        // refund dust eth, if any
        if (msg.value > amountETH) TransferHelper.safeTransferETH(msg.sender,
msg.value - amountETH);
```

However, the UniswapV2Migrator contract lacks a `receive()` function to actually receive those funds, meaning that such a call from the UniswapV2 Router will revert.
This is a DOS on the ability to add liquidity (in the form of native Ether) to the target Uniswap V2 Pool. An attacker can also send some Ether to the pool so as to create Dust.

Recommendations:
Add a `receive()` function to the UniswapV2Migrator contract.

Whetstone's response:

Fixed

H-05 Lack of access control and recipient validation in UniswapV2Migrator.migrate() can drain funds held by the contract

Severity: **High**	Impact: **Medium**	Likelihood: **High**
Files: UniswapV2Migrator.sol	Status: Fixed	Violated Property: N/A

Note: The Impact is medium due to the contract being supposed to only transiently hold funds.

Description

If `UniswapV2Migrator` holds any funds that aren't transferred atomically, any caller can input themselves as the recipients to recover them:

```
File: UniswapV2Migrator.sol
61:     function migrate(
62:         address token0,
63:         address token1,
64:         uint256 amount0,
65:         uint256 amount1,
66:         address recipient,
67:         bytes memory
68:     ) external payable returns (address pool, uint256 liquidity) {
//@audit-issue anyone can call
...
76:         if (token0 != address(0)) ERC20(token0).approve(address(router),
amount0);
77:         ERC20(token1).approve(address(router), amount1);
78:
79:         if (token0 == address(0)) {
80:             (,, liquidity) = router.addLiquidityETH{ value: amount0
}(token1, amount0, 0, 0, recipient, block.timestamp); //@audit-issue input
amount0 can be 0 and the next line will send the contract's balance to the
attacker
81:             SafeTransferLib.safeTransferETH(recipient,
```

```
address(this).balance);
82:         } else {
83:             ERC20(token0).approve(address(router), amount0);//@audit-issue
input amount0 can be 0
84:             (,, liquidity) = router.addLiquidity(token0, token1, 0, 0, 0,
0, msg.sender, block.timestamp); //@audit-issue no amount is sent to the
router anyway
85:             SafeTransferLib.safeTransfer(ERC20(token0), recipient,
ERC20(token0).balanceOf(address(this))); //@audit-issue all stored funds from
any token0 is sent to the attacker
86:         }
87:
88:         SafeTransferLib.safeTransfer(ERC20(token1), recipient,
ERC20(token1).balanceOf(address(this)));//@audit-issue all stored funds from
any token1 is sent to the attacker
89:     }
```

Recommendations:

Consider adding some access control so that only the Airlock contract can call `UniswapV2Migrator.migrate()`

Whetstone's response:

We don't think that's a problem, but these changes will prevent this from happening anyway: https://github.com/whetstoneresearch/doppler-smart-contracts/pull/213

H-06 UniswapV2Migrator.migrate() is griefable by frontrunning pool creations on Uniswap V2

Severity: **High**	Impact: **High**	Likelihood: **Medium**
Files: UniswapV2Migrator.sol	Status: Fixed	Violated Property: N/A

Note: The likelihood is medium due to the existence of a workaround.

Description

The `migrate()` function creates a pair on Uniswap V2 if it doesn't exist:

```
File: UniswapV2Migrator.sol
69:         pool = factory.getPair(token0, token1);
70:
71:         if (pool == address(0)) {
72:             pool =
73:                 factory.createPair(token0 == address(0) ?
0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2 : token0, token1);
74:         }
```

However, anyone can front-run the pair creation on Uniswap V2 and send some dust of 1 token to DOS, the call to `router.addLiquidity()`

Indeed, the expected scenario for a non-existing pool is for it to be empty when first calling `router.addLiquidity()`, so in UniswapV2Router02.sol#L32-L48 we'd be in the case of `reserveA == 0 && reserveB == 0`:

```
    // **** ADD LIQUIDITY ****
    function _addLiquidity(
        address tokenA,
```

```
        address tokenB,
        uint amountADesired,
        uint amountBDesired,
        uint amountAMin,
        uint amountBMin
    ) internal virtual returns (uint amountA, uint amountB) {
        // create the pair if it doesn't exist yet
        if (IUniswapV2Factory(factory).getPair(tokenA, tokenB) == address(0))
{
            IUniswapV2Factory(factory).createPair(tokenA, tokenB);
        }
        (uint reserveA, uint reserveB) = UniswapV2Library.getReserves(factory,
tokenA, tokenB);
        if (reserveA == 0 && reserveB == 0) {
            (amountA, amountB) = (amountADesired, amountBDesired);
        } else {
            uint amountBOptimal = UniswapV2Library.quote(amountADesired,
reserveA, reserveB);
```

In `UniswapV2Migrator.migrate()`, the call to `router.addLiquidity()` actually inputs 0 for the amounts, which is a bug in itself, but let's say that the bug is fixed and the correct amount would be input:

```
File: UniswapV2Migrator.sol
83:             ERC20(token0).approve(address(router), amount0);
- 84:             (,, liquidity) = router.addLiquidity(token0, token1, 0, 0,
0, 0, msg.sender, block.timestamp);
+ 84:             (,, liquidity) = router.addLiquidity(token0, token1,
amount0, amount1, 0, 0, msg.sender, block.timestamp);
```

If the Pool creation on Uniswap V2 was frontrun by an attacker sending dust funds in there, then instead of hitting the condition of `if (reserveA == 0 && reserveB == 0) {`, the code would actually call `UniswapV2Library.quote(amountADesired, reserveA, reserveB);`. It's implemented at UniswapV2Library.sol#L35-L40 as such:

```
    // given some amount of an asset and pair reserves, returns an equivalent
amount of the other asset
```

```
    function quote(uint amountA, uint reserveA, uint reserveB) internal pure
returns (uint amountB) {
        require(amountA > 0, 'UniswapV2Library: INSUFFICIENT_AMOUNT');
        require(reserveA > 0 && reserveB > 0, 'UniswapV2Library:
INSUFFICIENT_LIQUIDITY');
        amountB = amountA.mul(reserveB) / reserveA;
    }
```

As you may see, if one of the reserves is 0, then the transaction will revert, which is our scenario here.

Recommendations:

A workaround for this issue would be to forcibly send funds to the Uniswap v2 pool and forcibly synchronize balances and reserves (calling either UniswapV2Pair.skim() or UniswapV2Pair.sync()). This would make the migration possible.
However, to prevent the grief vector, this mechanism could be coded to send the absolute minimum of funds if the Pool exists and any of the reserves have a zero value.

Whetstone's response:

Fixed

H-07 In UniswapV2Migrator.migrate(), the call to router.addLiquidity() actually inputs 0 for the amounts

Severity: **High**	Impact: **Medium**	Likelihood: **High**
Files: UniswapV2Migrator.sol	Status: Fixed	Violated Property: N/A

Note: The impact is medium because the funds are not lost (they are expected to all be sent to the recipient).

Description

The `amount0` and `amount1` parameters are expected to be passed, but 0s are passed instead.

Recommendations:

```
File: UniswapV2Migrator.sol
83:             ERC20(token0).approve(address(router), amount0);
- 84:             (,, liquidity) = router.addLiquidity(token0, token1, 0, 0,
0, 0, msg.sender, block.timestamp);
+ 84:             (,, liquidity) = router.addLiquidity(token0, token1,
amount0, amount1, 0, 0, msg.sender, block.timestamp);
```

Whetstone's response:

Fixed

H-08 Token creation is frontrunnable and can grief users' calls to Airlock.create()

Severity: **High**	Impact: **Medium**	Likelihood: **High**
Files: Airlock.sol TokenFactory.sol	Status: Fixed	Violated Property: N/A

Description

`TokenFactory.create()` is publicly accessible and deploys using CREATE2:

```
File: TokenFactory.sol
07: contract TokenFactory is ITokenFactory {
08:     function create(
09:         string memory name,
10:         string memory symbol,
11:         uint256 initialSupply,
12:         address recipient,
13:         address owner,
14:         bytes memory,
15:         bytes32 salt
16:     ) external returns (address) {
17:         return address(new DERC20{ salt: salt }(name, symbol,
initialSupply, recipient, owner));
18:     }
19: }
20:
```

Given that the deployed addresses are computed as `new_address = hash(0xFF, sender, salt, bytecode)`: providing the `salt` by a frontrunning attack directly by calling `TokenFactory.create()` (mempool being monitored for transactions calling `Airlock.create()`) can grief the function at Airlock.sol#L107:

```
File: Airlock.sol
077:     function create(
...
107:         address token = tokenFactory.create(name, symbol, initialSupply,
address(this), address(this), tokenData, salt);
108:         address hook = hookFactory.create(poolManager, numTokensToSell,
hookData, salt);
```

Trying to create a token would revert.

These can be mitigated by ensuring that TokenFactory can only be called by Airlock.

Whetstone's response:

Fixed

H-09 Hook creation is frontrunnable and can grief users' call to Airlock.create()

Severity: **High**	Impact: **Medium**	Likelihood: **High**
Files: Airlock.sol DopplerFactory.sol	Status: Fixed	Violated Property: N/A

Description

`DopplerFactory.create()` is publicly accessible and deploys using CREATE2:

```
File: DopplerFactory.sol
09: contract DopplerFactory is IHookFactory {
10:     function create(
11:         IPoolManager poolManager,
12:         uint256 numTokensToSell,
13:         bytes memory data,
14:         bytes32 salt
15:     ) external returns (address) {
16:         (
17:             uint256 minimumProceeds,
18:             uint256 maximumProceeds,
19:             uint256 startingTime,
20:             uint256 endingTime,
21:             int24 startingTick,
22:             int24 endingTick,
23:             uint256 epochLength,
24:             int24 gamma,
25:             bool isToken0,
26:             uint256 numPDSlugs,
27:             address migrator
28:         ) = abi.decode(data, (uint256, uint256, uint256, uint256, int24,
int24, uint256, int24, bool, uint256, address));
29:
30:         return address(
31:             new Doppler{ salt: salt }(
```

```
32:                 poolManager,
33:                 numTokensToSell,
34:                 minimumProceeds,
35:                 maximumProceeds,
36:                 startingTime,
37:                 endingTime,
38:                 startingTick,
39:                 endingTick,
40:                 epochLength,
41:                 gamma,
42:                 isToken0,
43:                 numPDSlugs,
44:                 migrator
45:             )
46:         ); //@audit-info rename `address migrator` to `address airlock`
47:     }
48: }
```

Given that the deployed addresses are computed as `new_address = hash(0xFF, sender, salt, bytecode)`: providing the `salt` by a frontrunning attack directly by calling `DopplerFactory.create()` (mempool being monitored for transactions calling `Airlock.create()`) can grief the function at Airlock.sol#L108:

```
File: Airlock.sol
077:     function create(
...
107:         address token = tokenFactory.create(name, symbol, initialSupply,
address(this), address(this), tokenData, salt);
108:         address hook = hookFactory.create(poolManager, numTokensToSell,
hookData, salt);
```

Trying to create a token would revert.

These can be mitigated by ensuring that DopplerFactory can only be called by Airlock.

Whetstone's response:

https://github.com/whetstoneresearch/doppler-smart-contracts/pull/213

H-10 Airlock.create() is itself frontrunnable and can grief users' call to Airlock.create()

Severity: **High**	Impact: **Medium**	Likelihood: **High**
Files: Airlock.sol	Status: Fixed	Violated Property: N/A

Description

`Airlock.create()` is permissionless and front-runnable: as long as the right `salt` is provided, different parameters can be passed (like another owner, etc.). This will make rightful users' calls to `Airlock.create()` to revert.

This can be mitigated by deriving the salt from the key input parameters (making it so that a frontrunner would just be paying for the deployer's gas, so it wouldn't be a real attack anymore).

Whetstone's response:

Fixed

H-11 approve() may revert if the current approval is not zero

Severity: **High**	Impact: **High**	Likelihood: **Medium**
Files: UniswapV2Migrator.sol	Status: Fixed	Violated Property: N/A

Note: The likelihood is medium due to the existence of a workaround.

Description

- Some tokens (like the *very popular* USDT) do not work when changing the allowance from an existing non-zero allowance value (it will revert if the current approval is not zero to protect against front-running changes of approvals). These tokens must first be approved for zero, and then the actual allowance can be approved.
- Furthermore, OZ's implementation of safeApprove would throw an error if an approve is attempted from a non-zero value (`"SafeERC20: approve from non-zero to non-zero allowance"`)

Affected code:

- src/UniswapV2Migrator.sol

```
# File: src/UniswapV2Migrator.sol

UniswapV2Migrator.sol:76:         if (token0 != address(0))
ERC20(token0).approve(address(router), amount0);

UniswapV2Migrator.sol:77:         ERC20(token1).approve(address(router),
amount1);

UniswapV2Migrator.sol:83:             ERC20(token0).approve(address(router),
amount0);
```

A workaround exists as it's possible to directly call `UniswapV2Migrator.migrate()` with 0 amounts to reset the allowance

Recommendations:
Use OpenZeppelin's `forceApprove` function at SafeERC20.sol#L78-L94:

```
    /**
     * @dev Set the calling contract's allowance toward `spender` to `value`.
If `token` returns no value,
     * non-reverting calls are assumed to be successful. Meant to be used with
tokens that require the approval
     * to be set to zero before setting it to a non-zero value, such as USDT.
     *
     * NOTE: If the token implements ERC-7674, this function will not modify
any temporary allowance. This function
     * only sets the "standard" allowance. Any temporary allowance will remain
active, in addition to the value being
     * set here.
     */
    function forceApprove(IERC20 token, address spender, uint256 value)
internal {
        bytes memory approvalCall = abi.encodeCall(token.approve, (spender,
value));

        if (!_callOptionalReturnBool(token, approvalCall)) {
            _callOptionalReturn(token, abi.encodeCall(token.approve, (spender,
0)));
            _callOptionalReturn(token, approvalCall);
        }
    }
```

Whetstone's response:

Fixed

Medium Severity Issues

M-01 Lack of slippage protection in UniswapV2Migrator.migrate() risks value loss

Severity: **Medium**	Impact: **High**	Likelihood: **Low**
Files: UniswapV2Migrator.sol	Status: Fixed	Violated Property: N/A

Description

It is bad practice to have `amountAMin` and `amountBMin` set to 0, as this exposes the liquidity addition to sandwich attacks.

Recommendations:

```
File: UniswapV2Migrator.sol
- 80:             (,, liquidity) = router.addLiquidityETH{ value: amount0
}(token1, amount0, 0, 0, recipient, block.timestamp);
+ 80:             (,, liquidity) = router.addLiquidityETH{ value: amount0
}(token1, amount1, amountTokenMin, amountETHMin, recipient, block.timestamp);
...
- 84:             (,, liquidity) = router.addLiquidity(token0, token1, 0, 0,
0, 0, msg.sender, block.timestamp);
+ 84:             (,, liquidity) = router.addLiquidity(token0, token1,
amount0, amount1, amountAMin, amountBMin, recipient, block.timestamp);
```

Whetstone's response:

No liquidity will be in the pool (the pool is banned from holding any asset tokens beforehand). Since we're revoking the pool ban and we're the first to provide liquidity into it, we assume it's safe to skip the slippage protection.

certora

M-02 deadline == block.timestamp is ineffective

Severity: **Medium**	Impact: **Low**	Likelihood: **High**
Files: UniswapV2Migrator.sol	Status: Fixed	Violated Property: N/A

Description

Using `block.timestamp` used as a deadline effectively means no deadline (the transaction could wait for a year before being executed to move funds)

```
File: UniswapV2Migrator.sol
80:             (,, liquidity) = router.addLiquidityETH{ value: amount0
}(token1, amount0, 0, 0, recipient, block.timestamp);
...
84:             (,, liquidity) = router.addLiquidity(token0, token1, 0, 0, 0,
0, msg.sender, block.timestamp);
```

Recommendations:

Consider implementing an input parameter instead of using `block.timestamp`

Whetstone's response:

Fixed

We cannot use any other value since we cannot trust the address initializing the transaction (could be anyone) and we want the migration to be atomic and executed in this transaction.

M-03 Return value of transfer() not checked

Severity: **Medium**	Impact: **High**	Likelihood: **Low**
Files: Airlock.sol	Status: Fixed	Violated Property: N/A

Description

Not all `ERC20` implementations `revert()` when there's a failure in `transfer()`/`transferFrom()`. The function signature has a `boolean` return value and they indicate errors that way instead. By not checking the return value, operations that should have marked as failed, may potentially go through without actually making a payment

Affected code:

- src/Airlock.sol

```
# File: src/Airlock.sol

Airlock.sol:110:         ERC20(token).transfer(hook, numTokensToSell);

Airlock.sol:144:             ERC20(asset).transfer(tokenData.recipients[i],
tokenData.amounts[i]);

Airlock.sol:152:         if (currency0 != address(0))
ERC20(currency0).transfer(address(tokenData.migrator), amount0);

Airlock.sol:153:
ERC20(currency1).transfer(address(tokenData.migrator), amount1);
```

Recommendations:

Consider using `SafeTransferLib`'s `safeTransfer` just like it's done in the `UniswapV2Migrator` contract.

Whetstone's response:

Fixed token transfers will be handled directly by the Uniswap v4 currency library, which includes a "safe transfer" check.

Fix Review:

Indeed, Uniswap v4's currency library safely handles token transfers.

M-04 Incorrect loop start index in for positions in _unlockCallback()

Severity: **Medium**	Impact: **Low**	Likelihood: **High**
Files: Doppler.sol	Status: Fixed	Violated Property: N/A

Description

At Doppler.sol#L972-L973, the current loop starts from `salt == 0`:

```
File: Doppler.sol
972:             for (uint256 i; i < 3 + numPDSlugs; ++i) {
973:                 Position memory position =
positions[bytes32(uint256(i))];
```

As we have at most `NUM_DEFAULT_SLUGS - 1 + numPDSlugs` positions, the loop's limit is ok (`i < 3 + numPDSlugs`) but it should start at `salt == 1`.

Recommendations:

Consider starting from index 1. Also, it'd be great to replace the hardcoded 3 with `NUM_DEFAULT_SLUGS`:

```
- 972:             for (uint256 i; i < 3 + numPDSlugs; ++i) {
+ 972:             for (uint256 i = 1; i < NUM_DEFAULT_SLUGS + numPDSlugs;
++i) {
973:                 Position memory position =
positions[bytes32(uint256(i))];
```

Whetstone's response:

Fixed

Low Severity Issues

L-01 Some tokens may revert when zero value transfers are made

Severity: **Critical**	Impact: **Medium**	Likelihood: **Low**
Files: Airlock.sol UniswapV2Migrator.sol	Status: Fixed	Violated Property: N/A

Description

Example: https://github.com/d-xo/weird-erc20#revert-on-zero-value-transfers.

Although EIP-20 states that zero-valued transfers must be accepted, some tokens, such as LEND, will revert if this is attempted, which may cause transactions that involve other tokens (such as batch operations) to fully revert.

Affected code:

- src/Airlock.sol

```
# File: src/Airlock.sol

Airlock.sol:110:         ERC20(token).transfer(hook, numTokensToSell);

Airlock.sol:144:             ERC20(asset).transfer(tokenData.recipients[i],
tokenData.amounts[i]);

Airlock.sol:152:         if (currency0 != address(0))
ERC20(currency0).transfer(address(tokenData.migrator), amount0);

Airlock.sol:153:
ERC20(currency1).transfer(address(tokenData.migrator), amount1);
```

- src/UniswapV2Migrator.sol

```
# File: src/UniswapV2Migrator.sol

UniswapV2Migrator.sol:85:
SafeTransferLib.safeTransfer(ERC20(token0), recipient,
ERC20(token0).balanceOf(address(this)));

UniswapV2Migrator.sol:88:         SafeTransferLib.safeTransfer(ERC20(token1),
recipient, ERC20(token1).balanceOf(address(this)));
```

Recommendations:
Consider skipping the transfer if the amount is zero, which will also save gas.

Whetstone's response:

Fixed

Informational Severity Issues

I-01. The slugs array size returned by _computePriceDiscoverySlugsData() is too big

Description:

The returned array will have many empty slots

Recommendation:

The right size should be used:

```
File: Doppler.sol
771:     function _computePriceDiscoverySlugsData(
...
776:     ) internal view returns (SlugData[] memory) {
- 777:         SlugData[] memory slugs = new SlugData[](numPDSlugs);
...
798:         uint256 pdSlugsToLp = numPDSlugs;
799:         for (uint256 i = numPDSlugs; i > 0; --i) {
800:             if (_getEpochEndWithOffset(i - 1) !=
_getEpochEndWithOffset(i)) {
801:                 break;
802:             }
803:             --pdSlugsToLp;
804:         }
...
- 809:         for (uint256 i; i < numPDSlugs; ++i) {
+ 809:         SlugData[] memory slugs = new SlugData[](pdSlugsToLp);
+ 809:         for (uint256 i; i < pdSlugsToLp; ++i) {
```

Then the `_unlockCallback()` function will need to be adjusted to use the right array size too:

```
File: Doppler.sol
1011:             SlugData[] memory priceDiscoverySlugs =
1012:                 _computePriceDiscoverySlugsData(key, upperSlug,
tickUpper, assetRemaining);
1013:
- 1014:             Position[] memory newPositions = new
Position[](NUM_DEFAULT_SLUGS - 1 + numPDSlugs);
```

```
+ 1014:             Position[] memory newPositions = new
Position[](NUM_DEFAULT_SLUGS - 1 + priceDiscoverySlugs.length);
...
- 1041:             for (uint256 i; i < numPDSlugs; ++i) {
+ 1041:             for (uint256 i; i < priceDiscoverySlugs.length; ++i) {
```

Whetstone's response:

Fixed

certora

Disclaimer

Even though we hope this information is helpful, we provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the contract is secure in all dimensions. In no event shall Certora or any of its employees be liable for any claim, damages, or other liability, whether in an action of contract, tort, or otherwise, arising from, out of, or in connection with the results reported here.

About Certora

Certora is a Web3 security company that provides industry-leading formal verification tools and smart contract audits. Certora's flagship security product, Certora Prover, is a unique SaaS product that automatically locates even the most rare & hard-to-find bugs on your smart contracts or mathematically proves their absence. The Certora Prover plugs into your standard deployment pipeline. It is helpful for smart contract developers and security researchers during auditing and bug bounties.

Certora also provides services such as auditing, formal verification projects, and incident response.