Issue 1: Attacker can DOS depositNativeToken() and approveGenesisPool() permanently by creating a pair

Description:
Pair can be created via RouterV2.addLiquidity() if the caller has some amounts of both native and funding tokens.
Creating the pair and depositing funds to it would DOS depositNativeToken() and approveGenesisPool() permanently since these functions require both reserves to be 0.
Even if an attacker was not able to create pairs himself he would be able to DOS permanently approveGenesisPool() by sending 1 wei to the Pair and call sync() because there are require statements that need both pair reserves to be 0 on genesis pool approval.
There is still an edge case where he can perform the attack for depositNativeToken() but more requirements apply:
GenesisPoolManager.depositNativeToken() can be called with the same native/funding token that previously failed to launch a genesis pool successfully.
In this case a pair will already exist and code checks if the pool balance is empty:

```
None
require(IERC20(nativeToken).balanceOf(pairAddress) == 0, "!ZV");
            require(IERC20(_fundingToken).balanceOf(pairAddress) == 0,
"!ZV");
```

Attacker can send 1 wei to the pair address and DOS any future attempts for the manager to create a genesis pool for this exact pair of native/funding tokens.

Recommendation:
The fix that does not require big refactoring is allowing the pair reserves to be non-zero values across the life cycle of a genesis pool.
The downside is that there could be unused amounts of native or funding tokens when we add liquidity to the pair during a genesis pool launch because the pair could have small token amounts with undesired ratio between native and funding token.
To provide liquidity in the desired ratio the pair might need manual adjustment - transferring tokens and calling sync so that the desired pair ratio is reached in the same transaction as the pool launch.
To prevent frontrunning that could change the ratio yet again this should be done by a private transaction

---

Issue 2: Attacker can DOS genesis pool pre-launches by creating a gauge

Description:
An attacker can create a gauge by calling GaugeManager.createGauge() which will make _preLaunchPool() to revert because createGauge() reverts when a gauge for a specific pair has already been created.
The conditions for this to be possible are that both native and funding tokens have to be whitelisted in the TokenHandler.
The native token gets whitelisted when a genesis pool pre-launches.
When we already have one pre-launched pool for native token X and a new one is created, the newly created genesis pool's pre-launch can be DOSed by somebody creating a gauge for this pair.
It is not mandatory the for the native + new funding token Pair contract to be created via GenesisPoolManager.depositNativeToken() because anybody can create the Pair contract via RouterV2.addLiquidity() as long as he has some amounts of both tokens.

We assume that the native token will not be whitelisted manually by governance (which would allow the attacker to create a gauge for the pair right after pair creation which currently is on genesis pool creation) because TokenHandler.whitelistToken() reverts when called with already whitelisted token. If done this would have blocked pre-launches as they attempt to whitelist the native token each time.

Recommendation:
If there is already a deployed gauge use it instead of attempting to create a new one on pre-launch.

Issue 3: Genesis pool owner and deposit user funds can get locked in an edge case

Description:
The goal of genesis pools is to launch a native token when a certain amount of funding tokens get collected from users.
In case not enough funds are collected for a certain amount of time the pool status should be changed to NOT_QUALIFIED.
This status will allow users and pool owner to withdraw the funds they have deposited initially. Also this status will make the GenesisPoolManager logic to create a new genesis pool when GenesisPoolManager.depositNativeToken() gets called to attempt again to launch the native token.
If for any reason governance or a whitelisted address calls GenesisPoolManager.depositNativeToken() before the pool status is changed to NOT_QUALIFIED then the pool will get reconfigured - erasing key storage variables and any present funds will get locked with no way of retrieving them.
Notice that GenesisPoolManager.depositNativeToken() can get called after a genesis pool that has already launched with a different funding token. In this case the normal genesis pool contract logic would break.
Recommendation:
We recommend introducing a check in GenesisPool.setGenesisPoolInfo() that will make the call to revert if the function is called for a second time.
Furthermore, we want to make sure that previously failed pools remain untouched so depositors can retrieve their funds.
We can do this by changing in GenesisPoolManager.depositNativeToken() this part:

```
None
genesisPool = genesisFactory.getGenesisPool(nativeToken);
       if (genesisPool == address(0))
           genesisPool = genesisFactory.createGenesisPool(
               _sender,
               nativeToken,
               _fundingToken
           );
```

to check if a specific genesis pool made for nativeToken/fundingToken pair is with status != NOT_QUALIFIED or does not exist. If this is the case create a new genesis pool. If there is an active pool - revert.

Issue 4: Genesis pool launches can be temporarily stopped until owner does not take action
Description:
When enough funds are collected pool launches by adding liquidity to the corresponding nativeToken/fundingToken pair.
The pair is created at the same time the genesis pool was so this could mean many weeks.
At any point in time anybody can send some funding tokens (even 1 wei) to the pair and call the sync() function.
Upon launch this will cause a revert: GenesisPool.launch() -> RouterV2.addLiquidity() -> RouterV2._addLiquidity() -> RouterV2.quoteLiquidity() -> revert.

```
None
function _addLiquidity(
       address tokenA,
       address tokenB,
       bool stable,
       uint amountADesired,
       uint amountBDesired,
       uint amountAMin,
       uint amountBMin
   ) internal returns (uint amountA, uint amountB) {
       ...
       (uint reserveA, uint reserveB) = getReserves(tokenA, tokenB, stable);
       // @audit 1 of the reserves will be non zero due to the funding token
transfer + sync()
       if (reserveA == 0 && reserveB == 0) {
           (amountA, amountB) = (amountADesired, amountBDesired);
       } else {
@>         uint amountBOptimal = quoteLiquidity(
               amountADesired,
               reserveA,
               reserveB
           );
           if (amountBOptimal <= amountBDesired) {
               (amountA, amountB) = (amountADesired, amountBOptimal);
           } else {
               uint amountAOptimal = quoteLiquidity(
                   amountBDesired,
                   reserveB,
                   reserveA
               );
               assert(amountAOptimal <= amountADesired);
               (amountA, amountB) = (amountAOptimal, amountBDesired);
           }
       }
       require(amountA >= amountAMin && amountB >= amountBMin, "IAA");
   }
function quoteLiquidity(uint amountA, uint reserveA, uint reserveB) internal
pure returns (uint amountB) {
@>     require(amountA > 0 && reserveA > 0 && reserveB > 0, 'INL');
```

```
        amountB = amountA * reserveB / reserveA;
    }
```

The owner of the genesis pool will be forced to add tokens to the pair to launch the pool successfully. If the chain has a public mempool then he will have to do this with the help of private transactions since the attacker can keep frontrunning and depositing some amounts to the pair that could cause the pair ratio to change.
It is possible that this delay could make the pool expired.
Recommendation:
Same recommendation as issue 1.