RENASCENCE

MOR20 Audit Report

Version 2.0

Audited by:

HollaDieWaldfee

alexxander

June 13, 2024

Contents

1 Introduction **2**

1.1 About Renascence 2

1.2 Disclaimer 2

1.3 Risk Classification 2

2 Executive Summary **3**

2.1 About MOR20 3

2.2 Overview 3

2.3 Issues Found 3

3 Findings Summary **4**

4 Findings **5**

4.1 Centralization Risks 17

4.2 Systemic Risks 18

1 Introduction

1.1 About Renascence

Renascence Labs was established by a team of experts including HollaDieWaldfee, MiloTruck, alexxander and bytes032.

Our founders have a distinguished history of achieving top honors in competitive audit contests, enhancing the security of leading protocols such as Reserve Protocol, Arbitrum, MaiaDAO, Chainlink, Dodo, Lens Protocol, Wenwin, PartyDAO, Lukso, Perennial Finance, Mute and Taurus.

We strive to deliver tailored solutions by thoroughly understanding each client's unique challenges and requirements. Our approach goes beyond addressing immediate security concerns; we are dedicated to fostering the enduring success and growth of our partners.

More of our work can be found here.

1.2 Disclaimer

This report reflects an analysis conducted within a defined scope and time frame, based on provided materials and documentation. It does not encompass all possible vulnerabilities and should not be considered exhaustive.

The review and accompanying report are presented on an 'as-is' and 'as-available' basis, without any express or implied warranties.

Furthermore, this report neither endorses any specific project or team nor assures the complete security of the project.

1.3 Risk Classification

	Impact: High	**Impact: Medium**	**Impact: Low**
Likelihood: High	High	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

1.3.1 Impact

- High - Funds are **directly** at risk, or a **severe** disruption of the protocol’s core functionality
- Medium - Funds are **indirectly** at risk, or **some** disruption of the protocol’s functionality
- Low - Funds are **not** at risk

1.3.2 Likelihood

- High - almost certain to happen, easy to perform, or not easy but highly incentivized
- Medium - only conditionally possible or incentivized, but still relatively likely
- Low - requires stars to align, or little-to-no incentive

2 Executive Summary

2.1 About MOR20

MOR20 is the generalized version of the Morpheus Capital Smart Contracts available for community use. A project deployer re-creates the Morpheus Fair Launch and Distribution project through the MOR20 Factory contracts, configures the project's distribution settings, and appoints a project owner who can make further adjustments. On deployment, the project deployer can also decide to trust the Morpheus team with upgrades of the smart contracts or freeze the smart contracts at a given implementation. Trusting the Morpheus team with upgrades can also be managed by the project owner after the deployment of the project contracts.

2.2 Overview

Project	MOR20
Repository	MOR20
Commit Hash	64643c146854...
Mitigation Hash	ded6fe395764...
Date	30 May 2024 - 4 June 2024

2.3 Issues Found

Severity	**Count**
High Risk	0
Medium Risk	3
Low Risk	2
Informational	5
Total Issues	**10**

3 Findings Summary

ID	Description	Status
M-1	Parent contract `Factory.sol` is missing storage gap which could lead to storage collisions if it's upgraded	Resolved
M-2	Incorrect check in `Distribution.editPool()` allows modifying pool configurations and reward parameter changes can lock stakes	Resolved
M-3	The `Factory` administrator can upgrade frozen Proxies through upgrading the `Factory` implementation	Resolved
L-1	`FeeConfig.__FeeConfig_init()` is missing a `baseFee` constraint check	Resolved
L-2	Public functions in `FreezableBeaconProxy` can lead to function selector collision with implementation	Resolved
I-1	Ambiguity in the fee constraint in `FeeConfig.sol`	Resolved
I-2	`abi.encode()` should be used in `Factory._calculatePoolSalt()` to avoid hash collisions.	Acknowledged
I-3	`FeeConfig.sol` is missing a call to `_disableInitializers()` in the constructor	Resolved
I-4	Code Improvements	Resolved
I-5	`FeeConfig` is deployed behind a ERC1967 proxy but can't be upgraded	Resolved

4 Findings

Medium Risk

[M-1] Parent contract Factory.sol is missing storage gap which could lead to storage collisions if its upgraded

Context:

- Factory.sol#L16

Description: Upgradeable parent contracts must implement a storage gap to allow the addition of new state variables in the future without compromising the storage compatibility with existing deployments. Without a storage gap, if there are any new variables in the Factory.sol contract, they will override variables in child contracts such as L1Factory.sol and L2Factory.sol. **Recommendation:**

```
@@ -26,6 +26,7 @@ abstract contract Factory is IFactory, OwnableUpgradeable,
PausableUpgradeable,
     mapping(address deployer => mapping(string protocol => mapping(string poolType =>
     address))) private _proxyPools;
     mapping(address deployer => DynamicSet.StringSet) private _protocols;

+    uint256[46] __gap;
     function __Factory_init() internal onlyInitializing {}
```

Morpheus: Fixed.

Renascence: The recommendation has been implemented.

[M-2] Incorrect check in Distribution.editPool() allows modifying pool configurations and reward parameter changes can lock stakes

Context:

- Distribution.sol#L87

Description: The Distribution.editPool() function both incorrectly considers the new pool config, i.e., pool_, and performs an incorrect check > block.timestamp to determine if the Pool payout has started. This allows the modification of pool.payoutStart, pool.withdrawLockPeriod, and pool.withdrawLockPeriodAfterStake while the payout period has already started.

```
if (pool_.payoutStart > block.timestamp) {
    require(pool.payoutStart == pool_.payoutStart, ”DS: invalid payout start value”);
    require(pool.withdrawLockPeriod == pool_.withdrawLockPeriod, ”DS: invalid WLP
    value”);
    require(pool.withdrawLockPeriodAfterStake == pool_.withdrawLockPeriodAfterStake,
    ”DS: invalid WLPAS value”);
}
```

Another problem is that the remaining pool parameters, like initialReward, are not checked at all. It has been described in the centralization risks section of the previous Morpheus audit how this can also lead to stakes being locked.

The pool.initialReward variable for a given pool can be set by an admin through Distribution.editPool() to a large number such that a call to Distribution._getCurrentPoolRate() will revert because of an overflow in functions such as LinearDistributionIntervalDecrease.calculateMaxEndTime() and LinearDistributionIntervalDecrease.- calculateFullPeriodReward(). Since Distribution._getCurrentPoolRate() is invoked during Distribution.claim(), Distribution.stake(), Distribution.withdraw(), and Distribution.editPool() the user's funds can remain locked in the contract without the possibility for the pool to be edited back in a state that can recover the funds

Recommendation: The `Distribution.editPool()` function should check against the current pool configuration to determine if the payout period has started.

```
@@ -84,7 +84,7 @@ contract Distribution is IDistribution, OwnableUpgradeable {
         Pool storage pool = pools[poolId_];
         require(pool.isPublic == pool_.isPublic, ”DS: invalid pool type”);
-        if (pool_.payoutStart > block.timestamp) {
+        if (pool.payoutStart <= block.timestamp) {
```

It must also be determined which permissions exactly the `Distribution` owner should have. Providing sanity checks is a much weaker requirement than ensuring no stakes can be locked. If no stakes should be locked, then all pool parameters must be checked and by testing it must be ensured that changing the parameters to their limits cannot break calculations.

Morpheus: Fixed by removing the `Distribution.editPool()` function.

Renascence: By removing the `editPool()` function, users can rely on the parameters that a pool is created with. After the pool creation, the protocol owner cannot edit the pool and prevent users from withdrawing their `stETH`.

[M-3] The `Factory` administrator can upgrade frozen Proxies through upgrading the `Factory` implementation

Context:

- Factory.sol

Description: Currently, in a project created through Morpheus Factory contracts, the `Distribution`, `L1Sender`, `L2MessageReceiver`, and `L2TokenReceiver` contracts are deployed behind a `FreezableBeaconProxy`. The `FreezableBeaconProxy` allows the `Factory` to access the functions `FreezableBeaconProxy.freeze()` and `FreezableBeaconProxy.unfreeze()`, which are intended to allow the `project deployer` to opt in or out of Beacon upgrades. The current implementation of `Factory` allows only the `project deployer` of a project to call `Factory.freezePool()` and `Factory.unfreezePool()`.

However, the `Factory` administrator can upgrade `Factory` to an implementation that allows arbitrary access to `Factory.freezePool()` and `Factory.unfreezePool()`, thereby gaining access to any project's `FreezableBeaconProxy.freeze()` and `FreezableBeaconProxy.unfreeze()` functions. This means that any `project deployer`, regardless of whether they have their `FreezableBeaconProxy` frozen or not, can have their `Distribution`, `L1Sender`, `L2MessageReceiver`, and `L2TokenReceiver` contracts forced to undergo a Beacon upgrade to an arbitrary implementation set by the `Factory` administrator. **Recommendation:** A potential remedy would be to remove the upgradeability of the `Factory` contract. Another option to consider would be to allow the deployment owner to grant and revoke privileges from the factory, however, this alters the balance of privilege between the `project deployer` and `project owner`.

Here are the changes to remove upgradeability.

Changes to `Factory.sol`.

```
@@ -2,9 +2,9 @@
 pragma solidity ^0.8.20;

 import {Create2} from ”@openzeppelin/contracts/utils/Create2.sol”;
-import {UUPSUpgradeable} from
”@openzeppelin/contracts-upgradeable/proxy/utils/UUPSUpgradeable.sol”;
-import {OwnableUpgradeable} from
”@openzeppelin/contracts-upgradeable/access/OwnableUpgradeable.sol”;
-import {PausableUpgradeable} from
”@openzeppelin/contracts-upgradeable/security/PausableUpgradeable.sol”;
+import {Ownable} from ”@openzeppelin/contracts/access/Ownable.sol”;
+import {Pausable} from ”@openzeppelin/contracts/security/Pausable.sol”;
+
 import {UpgradeableBeacon} from
 ”@openzeppelin/contracts/proxy/beacon/UpgradeableBeacon.sol”;

 import {DynamicSet} from
 ”@solarity/solidity-lib/libs/data-structures/DynamicSet.sol”;
@@ -13,7 +13,7 @@ import {Paginator} from
”@solarity/solidity-lib/libs/arrays/Paginator.sol”;
 import {IFactory} from ”../interfaces/factories/IFactory.sol”;
 import {IFreezableBeaconProxy, FreezableBeaconProxy} from
 ”../proxy/FreezableBeaconProxy.sol”;

-abstract contract Factory is IFactory, OwnableUpgradeable, PausableUpgradeable,
UUPSUpgradeable {
+abstract contract Factory is IFactory, Ownable, Pausable {
     using DynamicSet for DynamicSet.StringSet;
     using Paginator for DynamicSet.StringSet;

@@ -26,8 +26,6 @@ abstract contract Factory is IFactory, OwnableUpgradeable,
PausableUpgradeable,
     mapping(address deployer => mapping(string protocol => mapping(string poolType =>
     address))) private _proxyPools;
     mapping(address deployer => DynamicSet.StringSet) private _protocols;

-    function __Factory_init() internal onlyInitializing {}
-
     /**
      * @notice Returns contract to normal state.
      */

-    function _authorizeUpgrade(address) internal view override onlyOwner {}
 }
```

Changes to `L1Factory.sol`.

```
@@ -19,16 +19,7 @@ contract L1Factory is IL1Factory, Factory {
     ArbExternalDeps public arbExternalDeps;
     LzExternalDeps public lzExternalDeps;

-    constructor() {
-        _disableInitializers();
-    }
-
-    function L1Factory_init() external initializer {
-        __Pausable_init();
-        __Ownable_init();
-        __UUPSUpgradeable_init();
-        __Factory_init();
-    }
+    constructor() {}

     function setDepositTokenExternalDeps(
         DepositTokenExternalDeps calldata depositTokenExternalDeps_
@@ -38,20 +29,17 @@ contract L1Factory is IL1Factory, Factory {

         depositTokenExternalDeps = depositTokenExternalDeps_;
     }
-
     function setLzExternalDeps(LzExternalDeps calldata lzExternalDeps_) external
     onlyOwner {
         require(lzExternalDeps_.endpoint != address(0), ”L1F: invalid LZ endpoint”);
         require(lzExternalDeps_.destinationChainId != 0, ”L1F: invalid chain ID”);

         lzExternalDeps = lzExternalDeps_;
     }
-
     function setArbExternalDeps(ArbExternalDeps calldata arbExternalDeps_) external
     onlyOwner {
         require(arbExternalDeps_.endpoint != address(0), ”L1F: invalid ARB
         endpoint”);

         arbExternalDeps = arbExternalDeps_;
     }
-
     function setFeeConfig(address feeConfig_) external onlyOwner {
         require(feeConfig_ != address(0), ”L1F: invalid fee config”);
```

Changes to L2Factory.sol.

```
@@ -19,16 +19,7 @@ contract L2Factory is IL2Factory, Factory {

     mapping(address deployer => mapping(string protocol => address)) private _mor20;

-    constructor() {
-        _disableInitializers();
-    }
-
-    function L2Factory_init() external initializer {
-        __Pausable_init();
-        __Ownable_init();
-        __UUPSUpgradeable_init();
-        __Factory_init();
-    }
+    constructor() {}

     function setLzExternalDeps(LzExternalDeps calldata lzExternalDeps_) external
     onlyOwner {
         require(lzExternalDeps_.endpoint != address(0), ”L2F: invalid LZ endpoint”);
@@ -37,7 +28,6 @@ contract L2Factory is IL2Factory, Factory {

         lzExternalDeps = lzExternalDeps_;
     }
-
     function setUniswapExternalDeps(UniswapExternalDeps calldata
     uniswapExternalDeps_) external onlyOwner {
         require(uniswapExternalDeps_.router != address(0), ”L2F: invalid UNI
         router”);
         require(uniswapExternalDeps_.nonfungiblePositionManager != address(0), ”L2F:
         invalid NPM”);
```

Changes to IL1Factory.sol.

```
@@ -73,7 +73,6 @@ interface IL1Factory {
     /**
      * The function that initializes the contract.
      */
-    function L1Factory_init() external;

     /**
      * The function to get fee config address.
```

Changes to `IL2Factory.sol`.

```
@@ -66,7 +66,6 @@ interface IL2Factory {
     /**
      * The function that initializes the contract.
      */
-    function L2Factory_init() external;

     /**
      * The function that sets the LZ external dependencies.
```

Morpheus: We agree with this problem. We decided to change the permissions check for calling freeze/unfreeze functions to solve it. Now FreezableBeaconProxy is fully responsible for this. It dynamically checks the current owner of the contract when calling functions. We realize the possible problems if the owner() function is not present and we take this under our control. Because of this, the Factory will remain a proxy and the corresponding functions will be removed.

Renascence: The factory can no longer bypass freezing of implementations. And the `FreezableBeaconProxy` correctly checks that the protocol owner has not frozen the implementation.

It is noteworthy that this change removes the ability to freeze the protocol deployment from the protocol deployer. Protocol deployers do no longer hold any privileges. Instead, protocol owners do.

Low Risk

[L-1] `FeeConfig.__FeeConfig_init()` is missing a `baseFee` constraint check

Context:

- FeeConfig.sol

Description: The functions `FeeConfig.setFee()` and `FeeConfig.setBaseFee()` enforce that the fee is less than `PRECISION`. However, the initializer function `FeeConfig.__FeeConfig_init()` does not. A check should also be performed during initialization to prevent a misconfiguration of the `baseFee`.
Recommendation:

```
@@ -16,6 +16,8 @@ contract FeeConfig is IFeeConfig, OwnableUpgradeable {
     function __FeeConfig_init(address treasury_, uint256 baseFee_) external
     initializer {
         __Ownable_init();

+        require(baseFee_ <= PRECISION, "FC: invalid base fee");
         treasury = treasury_;
         baseFee = baseFee_;
```

Morpheus: Fixed.

Renascence: The initializer function now performs the recommended check by calling `setBaseFee()` instead of setting the `baseFee` storage variable directly.

[L-2] Public functions in `FreezableBeaconProxy` can lead to function selector collision with implementation

Context:

- FreezableBeaconProxy.sol#L31-L60

Description: `FreezableBeaconProxy` extends OZ's `BeaconProxy` and adds functionality for the `_FACTORY` to freeze and unfreeze the implementation. The problem is that the new public functions can collide with functions in the implementation, such that users that want to interact with the implementation logic, instead execute the logic in the proxy.

The `_FACTORY` owner is trusted to not set an implementation that causes a collision, that's why the issue is "Low" severity. Still, it is highly encouraged to keep the `FreezableBeaconProxy` fully transparent.

Recommendation: It is recommended to adopt a pattern like in OZ's `TransparentUpgradeableProxy`, that allows the `_FACTORY` to call privileged functions, while preserving transparency for other users.

```
diff --git a/contracts/proxy/FreezableBeaconProxy.sol
b/contracts/proxy/FreezableBeaconProxy.sol
index aee4a70..f50dd96 100644
--- a/contracts/proxy/FreezableBeaconProxy.sol
+++ b/contracts/proxy/FreezableBeaconProxy.sol
@@ -11,11 +11,7 @@ import {IFreezableBeaconProxy} from
"../interfaces/proxy/IFreezableBeaconProxy.s
  * The FreezableBeaconProxy is a beacon proxy contract with freeze/unfreeze features.
  * When the FreezableBeaconProxy is being frozen, the actual implementation is stored
  in the storage slot.
  */
-contract FreezableBeaconProxy is IFreezableBeaconProxy, BeaconProxy, Context {
-    modifier onlyFactory() {
-        _onlyFactory();
-        _;
-    }
+contract FreezableBeaconProxy is BeaconProxy, Context {

     bytes32 private constant _FREEZABLE_BEACON_PROXY_SLOT =
     keccak256("freezable.beacon.proxy.slot");

@@ -25,11 +21,23 @@ contract FreezableBeaconProxy is IFreezableBeaconProxy,
BeaconProxy, Context {
         _FACTORY = _msgSender();
     }

+    function _fallback() internal virtual override {
+        if (msg.sender == _FACTORY) {
+            if (msg.sig == IFreezableBeaconProxy.freeze.selector) {
+                freeze();
+            } else if (msg.sig == IFreezableBeaconProxy.unfreeze.selector) {
+                unfreeze();
+            }
+        } else {
+            super._fallback();
+        }
+    }
+
     /**
      * The function to freeze the implementation.
      */
-    function freeze() external onlyFactory {
-        require(!isFrozen(), "FBP: already frozen");
+    function freeze() internal {
+        require(StorageSlot.getAddressSlot(_FREEZABLE_BEACON_PROXY_SLOT).value ==
address(0), "FBP: already frozen");

         StorageSlot.getAddressSlot(_FREEZABLE_BEACON_PROXY_SLOT).value =
         _implementation();
     }
@@ -37,37 +45,17 @@ contract FreezableBeaconProxy is IFreezableBeaconProxy,
BeaconProxy, Context {
     /**
      * The function to unfreeze the implementation.
      */
-    function unfreeze() external onlyFactory {
-        require(isFrozen(), "FBP: not frozen");
+    function unfreeze() internal {
+        require(StorageSlot.getAddressSlot(_FREEZABLE_BEACON_PROXY_SLOT).value !=
address(0), "FBP: not frozen");

         delete StorageSlot.getAddressSlot(_FREEZABLE_BEACON_PROXY_SLOT).value;
     }

-    /**
-     * The function to check if the implementation is frozen.
-     * @return The boolean value to indicating if the implementation is frozen.
-     */
-    function isFrozen() public view returns (bool) {
-        return StorageSlot.getAddressSlot(_FREEZABLE_BEACON_PROXY_SLOT).value !=
address(0);
-    }
-
-    /**
-     * The function to get the implementation.
-     * @return The implementation address.
-     */
-    function implementation() external view returns (address) {
-        return _implementation();
-    }
-
     function _implementation() internal view override returns (address) {
-        if (isFrozen()) {
+        if (StorageSlot.getAddressSlot(_FREEZABLE_BEACON_PROXY_SLOT).value !=
address(0)) {
             return StorageSlot.getAddressSlot(_FREEZABLE_BEACON_PROXY_SLOT).value;
         }

         return IBeacon(_getBeacon()).implementation();
     }
-
-    function _onlyFactory() internal view {
-        require(_msgSender() == _FACTORY, "FBP: not factory");
-    }
 }
```

If the `isFrozen()` and `implementation()` functions should be publicly available, they can be implemented in the `_FACTORY` either by retrieving the values from `FreezableBeaconProxy` by adding new internal functions and extending `_fallback()` or by storing the necessary information in `Factory`.

Morpheus: We decided to change this point partially, so as not to degrade the readability of the code. We have changed the function names to more specific names to decrease collision chances. Also, developers will check selectors for collision in the future.

Renascence: There are no function selector clashes currently. Therefore it is sufficient to check whether upgrades introduce such clashes, even though having public functions in the proxy is not best practice.

Informational

[I-1] Ambiguity in the fee constraint in `FeeConfig.sol`

Context:

- FeeConfig.sol#L24
- FeeConfig.sol#L36

Description: `FeeConfig.setFee()` enforces that the fee is less than or equal to `PRECISION`, while `FeeConfig.setBaseFee()` enforces that the fee is strictly less than `PRECISION`. For consistency, it is best practice to have both functions restrict the fee using the same boundary condition - either less than or equal to `PRECISION` or strictly less than `PRECISION`. **Recommendation:**

```
@@ -33,7 +35,7 @@ contract FeeConfig is IFeeConfig, OwnableUpgradeable {
     }

     function setBaseFee(uint256 baseFee_) external onlyOwner {
-        require(baseFee_ < PRECISION, ”FC: invalid base fee”);
+        require(baseFee_ <= PRECISION, ”FC: invalid base fee”);

         baseFee = baseFee_;
```

Morpheus: Fixed.

Renascence: The `setBaseFee()` function now checks for `<= PRECISION`. In addition, the same check is now applied in the initializer function by calling `setBaseFee()` instead of setting the `baseFee` storage variable directly.

[I-2] `abi.encode()` should be used in `Factory._calculatePoolSalt()` to avoid hash collisions.

Context:

- Factory.sol#L209

Description: `Factory._calculatePoolSalt()` uses the strings `protocol_` and `poolType_` to compute the salt for deployment. However, using `abi.encodePacked()` can lead to hash collisions depending on what strings are added as valid `poolType_`. Currently, there are no immediate security concerns, but preventing such hash collisions is preferable since the Factory contracts are upgradeable and could have extended logic in the future. **Recommendation:**

```
@@ -206,7 +206,7 @@ abstract contract Factory is IFactory, OwnableUpgradeable,
PausableUpgradeable,
         string memory protocol_,
         string memory poolType_
     ) internal pure returns (bytes32) {
-        return keccak256(abi.encodePacked(sender_, protocol_, poolType_));
+        return keccak256(abi.encode(sender_, protocol_, poolType_));
     }
```

Morpheus: Acknowledged. We decided not to change this point because the current implementation is protected against collision.

[I-3] `FeeConfig.sol` is missing a call to `_disableInitializers()` in the constructor

Context:

- FeeConfig.sol#L16

Description: The best practice in contracts that inherit from `Initializable` is to disable the initializers since if left uninitialized they can be invoked in the implementation contract by an attacker. For example, there is a past vulnerability disclosure that demonstrates how initializers getting called in the implementation can lead to contract takeover where the attacker can appoint an owner and would self-destruct the implementation, therefore, bricking the Proxy: OZ post-mortem. Although this issue has been fixed from OZ version 4.3.2 it's still best practice to call `Initializable._disableInitializers()` in a constructor in the implementation.

```
# Initializable.sol

* [CAUTION]
 * ====
 * Avoid leaving a contract uninitialized.
 *
 * An uninitialized contract can be taken over by an attacker. This applies to both a
 proxy and its implementation
 * contract, which may impact the proxy. To prevent the implementation contract from
 being used, you should invoke
 * the {_disableInitializers} function in the constructor to automatically lock it
 when it is deployed:
 *
```

Recommendation:

```
@@ -12,9 +12,13 @@ contract FeeConfig is IFeeConfig, OwnableUpgradeable {
     uint256 public baseFee;

     mapping(address => uint256) public fees;

+    constructor() {
+        _disableInitializers();
+    }
```

Morpheus: Fixed.

Renascence: The recommendation has been implemented.

[I-4] Code Improvements

Context:

- FeeConfig.sol#L16

Description: The name of `FeeConfig.__FeeConfig_init()` can be changed to `FeeConfig_init()` to maintain code consistency with the rest of the contracts.

Recommendation:

```
@@ -12,9 +12,11 @@ contract FeeConfig is IFeeConfig, OwnableUpgradeable {
     uint256 public baseFee;

     mapping(address => uint256) public fees;
-
-    function __FeeConfig_init(address treasury_, uint256 baseFee_) external initializer {
+
+    function FeeConfig_init(address treasury_, uint256 baseFee_) external initializer {
```

Morpheus: Fixed.

Renascence: The recommendation has been implemented.

[I-5] `FeeConfig` is deployed behind a ERC1967 proxy but cant be upgraded

Context:

- FeeConfig.sol#L10
- FeeConfig.test.ts#L27-L29

Description: In the test files, `FeeConfig` is deployed behind a ERC1967 proxy which needs to have its upgrade logic implemented in the implementation. However, `FeeConfig` does not inherit from `UUPSUpgradeable`, so it can't be upgraded.

Recommendation: It is possible to not deploy `FeeConfig` behind a proxy. Another option is to inherit from `UUPSUpgradeable` and make it upgradeable. Note that making `FeeConfig` upgradeable does not impact centralization concerns. Yes, it is possible that `FeeConfig` is made to revert but the outcome is equivalent to setting the fee percentage to 100% - in both cases no rewards are bridged to L2. It is important that `Distribution` compiles the call into `FeeConfig` as a staticcall, such that `FeeConfig` cannot reenter even when upgraded. Currently the call is compiled into staticcall because `getFeeAndTreasury()` is declared as `view`.

Morpheus: Fixed by using `UUPSUpgradeable`.

Renascence: `FeeConfig` now inherits from `UUPSUpgradeable` which fixes the issue as recommended.

4.1 Centralization Risks

4.1.1 Factory Administrator Must Be Trusted

The Factory administrator can change the implementation of any `FreezableBeaconProxy` to an arbitrary implementation if the project owner of the `FreezableBeaconProxy` hasn't opted out from Beacon upgrades through `FreezableBeaconProxy.freezeProxy_()`. The contracts deployed behind a `FreezableBeaconProxy` are `Distribution`, `L1Sender`, `L2MessageReceiver`, and `L2TokenReceiver`.

4.1.2 Fee Administrator Must Be Trusted

The fee administrator is the `owner` of `FeeConfig`. They have full control over the stETH earned from rebasings of any `MOR20` project deployed through the Factory contracts. However, the fee administrator cannot cause an impact on the staked stETH. Even by upgrading the `FeeConfig` implementation, the worst impact possible is that unclaimed yield gets lost which is the same as setting the fee percentage to 100%.

4.1.3 Project Owner Must Be Trusted

The owner of a `MOR20` project can use `FreezableBeaconProxy.freezeProxy_()` and `FreezableBeaconProxy.unfreezeProxy_()` during deployment or after deployment to opt in or out of a `FreezableBeaconProxy` for Beacon implementation upgrades managed by the Factory administrator. The contracts that the project owner can freeze or unfreeze are `Distribution`, `L1Sender`, `L2MessageReceiver`, and `L2TokenReceiver`. The project owner must be trusted to exercise due diligence, ensuring that any upgrades of the Beacon-provided implementation by the Factory are compatible with the rest of the contracts in a `MOR20` project.

Once set, the project owner is not able to change pool parameters in `Distribution`. This means that the project owner cannot interfere with users withdrawing their staked `stETH`. Unclaimed yield on the other hand can be affected by the project owner, who can prevent users from claiming their yield in various ways.

The project owner is in control of the generated staking yield in `Distribution.sol`. The project owner must be trusted to utilize the yield by initiating a cross-chain transfer through calling `Distribution.bridgeOverplus()` and managing the funds through the `L2TokenReceiver` contract on the destination chain.

The project owner can modify the `L1Sender` LayerZero config with arbitrary `zroPaymentAddress` and `adapterParams`. Configuring with invalid values can lead to `L1Sender.sendMintMessage()` reverting, causing users to be unable to claim their earned reward tokens.

The project owner can modify `config.sender` in `L2MessageReceiver`, potentially appointing a custom contract that can mint an arbitrary amount of the reward token.

The project owner can call `MOR20.updateMinter()` and enable an arbitrary contract to access the `MOR.mint()` function.

The project owner can withdraw all of the yield that is transferred to `L2TokenReceiver` by calling `L2TokenReceiver.withdrawToken()`. The project owner can also transfer any Uniswap Non-Fungible Position from `L2TokenReceiver` to an arbitrary address through `L2TokenReceiver.withdrawTokenId()`.

4.2 Systemic Risks

MOR20 deployments integrate with LayerZero, the Arbitrum bridge to bridge wstETH from Ethereum to Arbitrum and Uniswap V3. None of these integrations can prevent users from withdrawing their staked tokens according to the pool's parameters. The integrations can only interfere with reward payments.

On the other hand, Lido must be fully trusted as stETH is the token that is staked. If there is an issue in Lido, the staked funds are directly affected.