BENQI Ignite Audit Report

Prepared by Cyfrin

Version 2.0

Lead Auditors

Giovanni Di Siena

Immeas

December 11, 2024

Contents

1 About Cyfrin

Cyfrin is a Web3 security company dedicated to bringing industry-leading protection and education to our partners and their projects. Our goal is to create a safe, reliable, and transparent environment for everyone in Web3 and DeFi. Learn more about us at cyfrin.io.

2 Disclaimer

The Cyfrin team makes every effort to find as many vulnerabilities in the code as possible in the given time but holds no responsibility for the findings in this document. A security audit by the team does not endorse the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the solidity implementation of the contracts.

3 Risk Classification

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

4 Protocol Summary

Ignite by BENQI is a permissionless liquid staking protocol that subsidizes the stake requirement for validating on Avalanche. Zeeve is a partner validator hosting provider that assists in setting up and configuring nodes for Ignite users. Registrations made with hosted validators, paid in `QI`, are subject to additional `QI` rewards, proportional to the duration staked (one of 2, 4, 8, 12 weeks).

4.1 Actors & Roles

1. **Actors:**
    - **BENQI:** the liquid staking provider and creator of Ignite.
    - **Zeeve:** the partner validator hosting provider.
    - **Users:** the end-users of the system of Ignite contracts.
2. **Roles:**

- **Ignite:**
    - `Ignite::ROLE_WITHDRAW`: the role that allows `AVAX` to be withdrawn from the contract.
    - `Ignite::ROLE_REGISTER_WITHOUT_COLLATERAL`: the role that allows validators to be set up outside of the user-facing bounds of the system.
    - `Ignite::ROLE_RELEASE_LOCKED_TOKENS`: the role that can initiate the release of locked tokens.
    - `Ignite::ROLE_PAUSE`: the role that can pause contract functionality.
    - `Ignite::ROLE_UNPAUSE`: the role that can unpause contract functionality.
    - `Ignite::ROLE_REGISTER_WITH_FLEXIBLE_PRICE_CHECK`: the role granted to the staking contract to register a validator with pre-validated QI stakes.
    - `Ignite::DEFAULT_ADMIN_ROLE`: the role that can grant and revoke all other roles.

- **StakingContract:**
  - `StakingContract::BENQI_ADMIN_ROLE`: the role that allows BENQI to manage the staking contract, issue refunds, and set relevant parameters.
  - `StakingContract::BENQI_SUPER_ADMIN_ROLE`: the role that allows BENQI to manage the `BENQI_ADMIN_ROLE`.
  - `StakingContract::ZEEVE_ADMIN_ROLE`: the role that allows Zeeve to update the hosting fee and register nodes.
  - `StakingContract::ZEEVE_SUPER_ADMIN_ROLE`: the role that allows Zeeve to manage the `ZEEVE_ADMIN_ROLE`.
  - `StakingContract::DEFAULT_ADMIN_ROLE`: the role that can pause/unpause contract functionality and (inadvertently) grant/revoke all other roles.
- **ValidatorRewarder:**
  - `ValidatorRewarder::ROLE_WITHDRAW`: the role that can withdraw `QI` from the contract.
  - `ValidatorRewarder::ROLE_PAUSE`: the role that can pause contract functionality.
  - `ValidatorRewarder::ROLE_UNPAUSE`: the role that can unpause contract functionality.
  - `ValidatorRewarder::DEFAULT_ADMIN_ROLE`: the role that can grant and revoke all other roles.

4.2 Key Components

1. **Ignite:** the primary contract which manages staking registrations and rewards distribution.
2. **StakingContract:** the secondary contract responsible for managing pre-validated `QI` stakes and registering hosted nodes.
3. **ValidatorRewarder:** the contract responsible for distributing `QI` rewards to stakers using hosted nodes.
4. **Off-Chain Service:** the peripheral infrastructure responsible for performing BLS proof validation, depositing `AVAX` to BENQI Liquid Staked AVAX (sAVAX) on the Avalanche P-Chain, releasing locked tokens, etc.

4.3 Ignite Flow

1. **Staking & Registration:**
   - **Pay As You Go (PAYG):** a pre-configured node can be registered with Ignite by paying a non-refundable one-time fee in either `AVAX` or one of the accepted ERC20 fee payment tokens (with a `5%` discount if paid in `QI`). The entire `2000 AVAX` validator stake requirement is subsidized by BENQI.
   - **Stake AVAX:** a pre-configured node can be registered with Ignite by directly staking an amount of `AVAX` between the specified bounds, with the remaining amount subsidized by BENQI. No fees apply, although `10%` of the subsidized `AVAX` amount must also be paid in `QI`.
   - **Failed Registration:** a minimum contract balance is maintained to handle the case where registration fails, allowing the user to redeem their stake or registration fee.
2. **sAVAX Deposit:** staked `AVAX` is withdrawn by a privileged actor and deposited to BENQI Liquid Staked AVAX (sAVAX) on the Avalanche P-Chain.
3. **Validate:** the registered node validates for the specified duration and is subject to slashing in the event of excessive downtime.
4. **Reward/Slash/Release:** an amount of `AVAX` is withdrawn and returned to the contract for release by a privileged actor. If the node is slashed, less `AVAX` is returned to the contract; otherwise, the full amount plus any rewards are returned.
5. **Redeem:** the user can redeem their staked `AVAX` and any rewards after the validation period has ended and the funds are released.

4.4 StakingContract Flow

1. **Staking:**
   - **Stake AVAX:** a hosted node can be registered by paying the `200 AVAX` stake requirement, non-refundable `1 AVAX` Ignite fee, and non-refundable Zeeve hosting fee. The `201 AVAX` stake amount is swapped for `QI` for registration with Ignite and the entire `2000 AVAX` validator stake requirement is subsidized by BENQI.
   - **Stake ERC20:** a hosted node can be registered by paying the equivalent of the `201 AVAX` stake requirement and Zeeve hosting fee in one of the accepted ERC20 tokens. Again, the stake amount is swapped for `QI` for registration with Ignite and the entire `2000 AVAX` validator stake requirement is subsidized by BENQI.
   - **Refund Stake:** a refund can be requested to be executed by the BENQI admin, within the specified refund cooldown period, so long as a validator has not yet been provisioned.
2. **Registration:**
   - **Ignite Registration:** once the validator is configured by Zeeve, the node is registered with Ignite using the pre-validated `QI` stake. The Ignite fee is recalculated based on the `QI` amount and sent to the fee receipient.
   - **Failed Registration:** if registration fails, for example if the BLS proof is invalid, the funds are released by a privileged actor and the user can redeem their stake in `QI`.
3. **sAVAX Deposit:** staked `AVAX` is withdrawn by a privileged actor and deposited to BENQI Liquid Staked AVAX (sAVAX) on the Avalanche P-Chain.
4. **Validate:** the registered node validates for the specified duration, but pre-validated `QI` stakes are not subject to slashing.
5. **Reward/Release:** hosted validators are always eligible for `QI` rewards, calculated by `ValidatorRewarder`, proportional to the duration staked.
6. **Redeem:** the user can redeem their staked `QI` and additional rewards after the validation period has ended and the funds are released.

5 Audit Scope

Cyfrin conducted an audit of BENQI Ignite based on the code present in the repository commit hash bbca0dd, including and additional logic provided by Zeeve at commit hash b633362.

The following contracts were included in the scope of the audit:

```
- Ignite.sol
- IgniteStorage.sol
- staking.sol
- ValidatorRewarder.sol
```

6 Executive Summary

Over the course of 9 days, the Cyfrin team conducted an audit on the BENQI Ignite smart contracts provided by BENQI. In this period, a total of 30 issues were found.

This review of Ignite and the additional staking contracts yielded one high severity vulnerability that arose due to a failure to consider `QI` reward eligibility when refunding failed registrations. This oversight could have resulted in the complete draining of the `ValidatorRewarder` contract, and so should be addressed as a priority.

Five medium severity issues were also identified, relating to assumptions about native token transfers, low-level calls, privileged roles, failed registration, and an incorrect state update. These issues are less immediately severe but are still strongly recommended to be addressed to avoid potential loss of user funds and denial-of-service, along with the additional low and informational findings.

The Hardhat test suites cover the main functionalities of the contracts, including basic testing of both happy and unhappy paths; however, there is a notable lack of integration tests between the Ignite and staking contracts. Therefore, as part of this review, a Foundry test suite was developed to facilitate more comprehensive testing of all the Ignite contracts using various test fixtures and techniques such fork testing.

Considering the number of issues identified, it is statistically likely that there are more complex bugs still present that could not be identified given the time-boxed nature of this engagement. Due to the number of issues identified, the non-trivial changes required during mitigation, and the short turnaround time for reviewing the mitigation fixes, it is recommended that a competitive audit be undertaken prior to deploying significant monetary capital to production.

Summary

Project Name	BENQI Ignite
Repositories	ignite-contracts; benqi_smartcontract
Commits	bbca0ddb3992. . . ; b63336201f50. . .
Audit Timeline	Sep 16th - Sep 26th
Methods	Manual Review, Fork Testing

Issues Found

Critical Risk	0
High Risk	1
Medium Risk	5
Low Risk	5
Informational	14
Gas Optimizations	5
Total Issues	30

Summary of Findings

[H-1] Zeeve admin could drain `ValidatorRewarder` by abusing off-chain BLS validation due to `QI` rewards being granted to failed registrations	Resolved
[M-1] Redemption of slashed registrations could result in DoS due to incorrect state update	Resolved
[M-2] The default admin role controls all other roles within `StakingContract`	Resolved
[M-3] Inconsistent transfers of native tokens could result in unexpected loss of funds	Resolved

[M-4] Redemption of failed registration fees and pre-validated QI is not guaranteed to be possible	Resolved
[M-5] Ignite fee is not returned for pre-validated `QI` stakes in the event of registration failure	Resolved
[L-1] `StakingContract` refunds are affected by global parameter updates	Acknowledged
[L-2] Insufficient validation of Chainlink price feeds	Acknowledged
[L-3] Incorrect operator when validating subsidisation cap	Resolved
[L-4] `StakingContract::slippage` can be outside of `minSlippage` or `maxSlippage`	Resolved
[L-5] Lack of user-defined slippage and deadline parameters in `StakingContract::swapForQI` may result in unfavorable `QI` token swaps	Acknowledged
[I-01] `AccessControlUpgradeable::_setupRole` is deprecated	Resolved
[I-02] Unchained initializers should be called instead	Resolved
[I-03] Missing `onlyInitializing` modifier in `StakingContract`	Resolved
[I-04] Unnecessary `amount` parameter in `StakingContract::stakeWithERC20`	Acknowledged
[I-05] Staking amount in QI should be calculated differently	Acknowledged
[I-06] Tokens with more than `18` decimals will not be supported	Acknowledged
[I-07] Incorrect revert strings in `StakingContract::revokeAdminRole`	Resolved
[I-08] Placeholder recipient constants in `Ignite` should be updated before deployment	Acknowledged
[I-09] Missing modifiers	Resolved
[I-10] Incorrect assumption that Chainlink price feeds will always have the same decimals	Acknowledged
[I-11] Typo in `Ignite::registerWithPrevalidatedQiStake` NatSpec	Resolved
[I-12] Magic numbers should be replaced by constant variables	Acknowledged
[I-13] Misalignment of `pause()` and `unpause()` access controls across contracts	Acknowledged
[I-14] Inconsistent price validation in `Ignite::registerWithStake`	Acknowledged
[G-1] Unnecessary validation of `EnumerableSet` functions	Resolved
[G-2] Unnecessary validation in `StakingContract::registerNode`	Acknowledged
[G-3] Unnecessary conditional block in `Ignite::getTotalRegistrations` can removed	Resolved
[G-4] Unnecessary validation in `Ignite::getRegistrationsByAccount`	Resolved
[G-5] Unnecessary price feed address validation in `Ignite::configurePriceFeed`	Resolved

7 Findings

7.1 High Risk

7.1.1 Zeeve admin could drain `ValidatorRewarder` by abusing off-chain BLS validation due to `QI` rewards being granted to failed registrations

Description: `Ignite::releaseLockedTokens` takes the `failed` boolean as a parameter, intended to indicate that registration of the node has failed and allow the user's stake to be recovered by a call to `Ignite::redeemAfterExpiry`. Registrations made by `Ignite::registerWithAvaxFee` and `Ignite::registerWithErc20Fee` are handled within the first conditional branch; however, those made via `Ignite::registerWithStake` and `Ignite::registerWithPrevalidatedQiStake` are not considered until the final conditional branch shown below:

```
} else {
    avaxRedemptionAmount = avaxDepositAmount + registration.rewardAmount;
    qiRedemptionAmount = qiDepositAmount;

    if (qiRewardEligibilityByNodeId[nodeId]) {
        qiRedemptionAmount += validatorRewarder.claimRewards(
            registration.validationDuration,
            registration.tokenDeposits.tokenAmount
        );
    }

    minimumContractBalance -= avaxRedemptionAmount;
}
```

This is fine for registrations made with `AVAX` stake, since the reward amount is never updated from `0`; however, for those made with pre-validated QI stake, the call to `ValidatorRewarder::claimRewards` is executed regardless, returning the original stake in QI plus QI rewards for the full duration.

Furthermore, this behavior could be abused by the Zeeve admin to drain `QI` tokens from the `ValidatorRewarder` contract. Assuming interactions are made directly with the deployed contracts to bypass frontend checks, a faulty BLS proof can be provided to `StakingContract::registerNode` – this BLS proof is validated by an off-chain service when the `NewRegistration` event is detected, and `Ignite::releaseLockedTokens` will be called if it is invalid.

While Zeeve is somewhat of a trusted entity, they could very easily and relatively inconspicuously stake with burner user addresses, forcing the failure of BLS proof validation to drain the `ValidatorRewarder` contract due to the behavior of this off-chain logic in conjunction with the incorrect handling of `QI` rewards for failed registrations.

Impact: `QI` rewards will be paid to users of failed registrations made via `Ignite::registerWithPrevalidatedQiStake`. If abused by the Zeeve admin, then entire `ValidatorRewarder` contract balance could be drained.

Proof of Concept: The following test can be added to `Ignite.test.js` under `describe("Superpools")`:

```
it("earns qi rewards for failed registrations", async function () {
  await validatorRewarder.setTargetApr(1000);
  await ignite.setValidatorRewarder(validatorRewarder.address);
  await grantRole("ROLE_RELEASE_LOCKED_TOKENS", admin.address);

  // AVAX $20, QI $0.01
  const qiStake = hre.ethers.utils.parseEther("200").mul(2_000);
  const qiFee = hre.ethers.utils.parseEther("1").mul(2_000);

  // approve Ignite to spend pre-validated QI (bypassing StakingContract)
  await qi.approve(ignite.address, qiStake.add(qiFee));
  await ignite.registerWithPrevalidatedQiStake(
    admin.address,
    "NodeID-Superpools1",
    "0x" + blsPoP.toString("hex"),
    86400 * 28,
    qiStake.add(qiFee),
  );

  // registration of node fails
  await ignite.releaseLockedTokens("NodeID-Superpools1", true);

  const balanceBefore = await qi.balanceOf(admin.address);
  await ignite.connect(admin).redeemAfterExpiry("NodeID-Superpools1");

  const balanceAfter = await qi.balanceOf(admin.address);

  // stake + rewards are returned to the user
  expect(Number(balanceAfter.sub(balanceBefore))).to.be.greaterThan(Number(qiStake));
});
```

Recommended Mitigation: Avoid paying QI rewards to failed registrations by resetting the `qiRewardEligibilityByNodeId` state in `Ignite::releaseLockedTokens`:

```
    } else {
        minimumContractBalance += msg.value;
        totalSubsidisedAmount -= 2000e18 - msg.value;
+       qiRewardEligibilityByNodeId[nodeId] = false;
    }
```

BENQI: Fixed in commit 0255923.

Cyfrin: Verified, `QI` rewards are no longer granted to failed registrations.

7.2 Medium Risk

7.2.1 Redemption of slashed registrations could result in DoS due to incorrect state update

Description: Due to the logic surrounding Pay As You Go (PAYG) registrations, potential refunds associated with failed validators, and post-expiry redemptions of stake, a minimum balance of `AVAX` is required to remain within the `Ignite` contract. The `IgniteStorage::minimumContractBalance` state variable is responsible for keeping track of this balance and ensuring that `Ignite::withdraw` transfers the appropriate amount of `AVAX` to start validation.

When the validation period for a given registration has expired, `Ignite::releaseLockedTokens` is called by the privileged `ROLE_RELEASE_LOCKED_TOKENS` actor along with the redeemable tokens. If the registration is slashed, `minimumContractBalance` is updated to include `msg.value` less the slashed amount. Finally, when `Ignite::redeemAfterExpiry` is called by the original registerer to redeem the tokens, the `minimumContractBalance` state is again updated to discount this withdrawn amount.

However, this state update is incorrect as it should decrement the `avaxRedemptionAmount` rather than the `avaxDepositAmount` (which includes the already-accounted-for slashed amount). Therefore, `minimumContractBalance` will be smaller than intended, resulting in redemptions reverting due to underflow of the decrement or if a call to `Ignite::withdraw` leaves the contract with insufficient balance to fulfill its obligations.

Impact: Redemptions could revert if the current redemption is slashed and the state update underflows or if an earlier redemption is slashed and more `AVAX` is withdrawn than intended.

Proof of Concept: The following test can be added under `describe("users can withdraw tokens after the registration becomes withdrawable")` in `ignite.test.js`:

```
it("with slashing using a slash percentage", async function () {
  // Add AVAX slashing percentage to trigger the bug (50% so the numbers are easy)
  await ignite.setAvaxSlashPercentage("5000");

  // Register NodeID-1 for two weeks with 1000 AVAX and 200k QI
  await ignite.registerWithStake("NodeID-1", blsPoP, 86400 * 14, {
    value: hre.ethers.utils.parseEther("1000"),
  });

  // Release the registration with `msg.value` equal to AVAX deposit amount to trigger slashing
  await ignite.releaseLockedTokens("NodeID-1", false, {
    value: hre.ethers.utils.parseEther("1000"),
  });

  // The slashed amount is decremented from the minumum contract balance
  expect(await ignite.minimumContractBalance()).to.equal(hre.ethers.utils.parseEther("500"));

  // Reverts on underflow since it tries to subtract 1000 (avaxDepositAmount) from 500
  ↪   (minimumContractBalance)
  await expect(ignite.redeemAfterExpiry("NodeID-1")).to.be.reverted;
});
```

Note that this is not an issue for the existing deployed version of this contract as the AVAX slash percentage is zero.

Recommended Mitigation: Decrement `minimumContractBalance` by `avaxRedemptionAmount` instead of `avaxDepositAmount`:

```
if (registration.slashed) {
    avaxRedemptionAmount = avaxDepositAmount - avaxDepositAmount * registration.avaxSlashPercentage /
    ↪   10_000;
    qiRedemptionAmount = qiDepositAmount - qiDepositAmount * registration.qiSlashPercentage / 10_000;

-   minimumContractBalance -= avaxDepositAmount;
+   minimumContractBalance -= avaxRedemptionAmount;
```

```
} else {
```

BENQI: Fixed in commit fb686b8.

Cyfrin: Verified. The correct state update is now applied.

7.2.2 The default admin role controls all other roles within `StakingContract`

Description: Within `StakingContract`, there is intended separation between the Zeeve and BENQI admin/super-admin roles as implemented in the `grantAdminRole()`, `revokeAdminRole()`, and `updateAdminRole()` functions. The intention is for admin roles to be managed by the corresponding super-admin; however, `AccessControlUpgradeable::_setRoleAdmin` is never invoked for any of the roles and the current implementation fails to consider the default admin role that is granted to the BENQI super-admin for pausing purposes when the contract is initialized. As a result, the BENQI super-admin can be used to manage all other roles by invoking `AccessControlUpgradeable::grantRole` and `AccessControlUpgradeable::revokeRole` directly. This behavior is used in `Ignite` and `ValidatorRewarder` to grant the appropriate roles; however, it is not desirable in `StakingContract`.

Impact: The default admin role granted to the BENQI super-admin can be used to control all other roles.

Proof of Concept: The following test can be added to `stakingContract.test.js` under `describe("updateAdminRole")`:

```
it("allows BENQI_SUPER_ADMIN to update ZEEVE_SUPER_ADMIN", async function () {
    const zeeveSuperAdminRole = await stakingContract.ZEEVE_SUPER_ADMIN_ROLE();

    // BENQI_SUPER_ADMIN can use OpenZepplin's grantRole to alter ZEEVE_SUPER_ADMIN_ROLE
    await stakingContract.connect(benqiSuperAdmin).grantRole(zeeveSuperAdminRole, otherUser.address);
    await stakingContract.connect(benqiSuperAdmin).revokeRole(zeeveSuperAdminRole,
    ↪  zeeveSuperAdmin.address);

    expect(await stakingContract.hasRole(zeeveSuperAdminRole, otherUser.address)).to.be.true;
    expect(await stakingContract.hasRole(zeeveSuperAdminRole, zeeveSuperAdmin.address)).to.be.false;
});
```

An equivalent Foundry test can be run with the provided fixtures:

```
function test_defaultAdminControlsAllRolesPoC() public {
    assertEq(stakingContract.getRoleAdmin(stakingContract.DEFAULT_ADMIN_ROLE()),
    ↪  stakingContract.DEFAULT_ADMIN_ROLE());
    assertEq(stakingContract.getRoleAdmin(stakingContract.BENQI_SUPER_ADMIN_ROLE()),
    ↪  stakingContract.DEFAULT_ADMIN_ROLE());
    assertEq(stakingContract.getRoleAdmin(stakingContract.BENQI_ADMIN_ROLE()),
    ↪  stakingContract.DEFAULT_ADMIN_ROLE());
    assertEq(stakingContract.getRoleAdmin(stakingContract.ZEEVE_SUPER_ADMIN_ROLE()),
    ↪  stakingContract.DEFAULT_ADMIN_ROLE());
    assertEq(stakingContract.getRoleAdmin(stakingContract.ZEEVE_ADMIN_ROLE()),
    ↪  stakingContract.DEFAULT_ADMIN_ROLE());

    address EXTERNAL_ADDRESS = makeAddr("EXTERNAL_ADDRESS");
    vm.startPrank(BENQI_SUPER_ADMIN);
    stakingContract.grantRole(stakingContract.ZEEVE_SUPER_ADMIN_ROLE(), EXTERNAL_ADDRESS);
    vm.stopPrank();
    assertTrue(stakingContract.hasRole(stakingContract.ZEEVE_SUPER_ADMIN_ROLE(), EXTERNAL_ADDRESS));
}
```

Recommended Mitigation: Consider either:

1. Setting the appropriate role admins during initialization.

2. Removing the default admin role and creating a separate pauser role.
3. Overriding the OpenZeppelin functions to prevent them from being called directly.

BENQI: Fixed in commit 491a278.

Cyfrin: Verified. The OpenZeppelin function have been overridden.

7.2.3 Inconsistent transfers of native tokens could result in unexpected loss of funds

Description: Multiple protocol functions across both `Ignite` and `StakingContract` transfer native `AVAX` to the user and/or protocol fee/slashed token recipients.

Throughout `Ignite` [1, 2, 3, 4, 5, 6], the low-level `.call()` pattern is used; however, this same behavior is not followed in `StakingContract` – the `.transfer()` function is used on lines 433 and 484, and on line 728 `_transferETHAndWrapIfFailWithGasLimit()` is used, all with a `2300` gas stipend.

While these other functions may have been used to mitigate against potential re-entrancy attacks, native token transfers using low-level calls are preferred over `.transfer()` to mitigate against changes in gas costs, as described here. The instances on lines 433 and 484 are particularly problematic as they have no `WAVAX` fallback and could result in an unexpected loss of funds as described here and in the linked examples.

Impact: There could be an unexpected loss of funds if the recipient of a transfer (applicable to both users and the Zeeve wallet) is a smart contract that fails to implement a payable fallback function, or the fallback function uses more than 2300 gas units. This could happen, for example, if the recipient is a smart account whose fallback function logic causes the execution to use more than 2300 gas.

Recommended Mitigation: Consider modifying the instances of native token transfers in `StakingContract` to use low-level calls, making the necessary adjustments to protect against re-entrancy.

BENQI: Fixed in commit ee98629.

Cyfrin: Verified. The `_transferETHAndWrapIfFailWithGasLimit()` function is now used throughout `StakingContract`.

7.2.4 Redemption of failed registration fees and pre-validated QI is not guaranteed to be possible

Description: `Ignite::registerWithStake` performs a low-level call as part of its validation to ensure the beneficiary,in this case `msg.sender`, can receive `AVAX`:

```
// Verify that the sender can receive AVAX
(bool success, ) = msg.sender.call("");
require(success);
```

However, this is missing from `Ignite::registerWithAvaxFee`, meaning that failed registration fees are not guaranteed to be redeemable if the sender is a contract that cannot receive `AVAX`.

Similarly, `Ignite::registerWithPrevalidatedQiStake` performs no such validation on the beneficiary. While this may not seem to be problematic, since the stake requirement is provided in `QI`, there is a low-level call in `Ignite::redeemAfterExpiry` that will attempt a zero-value transfer for pre-validated `QI` stakes:

```
(bool success, ) = msg.sender.call{ value: avaxRedemptionAmount}("");
require(success);
```

If the specified beneficiary is a contract without a payable fallback/receive function then this call will fail. Furthermore, if this beneficiary contract is immutable, the `QI` stake will be locked in the `Ignite` contract unless it is upgraded.

Impact: Failed `AVAX` registration fees and prevalidated `QI` stakes will remain locked in the `Ignite` contract.

Proof of Concept: The following standalone Forge test demonstrates the behavior described above:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.15;

import "forge-std/Test.sol";

contract A {}

contract TestPayable is Test {
    address eoa;
    A a;

    function setUp() public {
        eoa = makeAddr("EOA");
        a = new A();
    }

    function test_payable() external {
        // Attempt to call an EOA with zero-value transfer
        (bool success, ) = eoa.call{value: 0 ether}("");

        // Assert that the call succeeded
        assertEq(success, true);

        // Attempt to call a contract that does not have a payable fallback/receive function with
        ↪  zero-value transfer
        (success, ) = address(a).call{value: 0 ether}("");

        // Assert that the call failed
        assertEq(success, false);
    }
}
```

Recommended Mitigation: Consider adding validation to `Ignite::registerWithAvaxFee` and `Ignite::registerWithPrevalidatedQiStake`. If performing a low-level call within `Ignite::registerWithPrevalidatedQiStake`, also consider adding the `nonReentrant` modifier.

BENQI: Fixed in commit 7d45908. There will no longer be a native token transfer for pre-validated QI stake registrations since this non-zero check is added before the call in commit f671224.

Cyfrin: Verified. The low-level call has been added to `Ignite::registerWithAvaxFee` and pre-validated QI stake registrations no longer have a zero-value call on redemption.

7.2.5 Ignite fee is not returned for pre-validated `QI` stakes in the event of registration failure

Description: The `1 AVAX` Ignite fee applied to pre-validated `QI` stakes is paid to the fee recipient at the time of registration. If this registration fails (e.g. due to off-chain BLS proof validation), the registration will be marked as withdrawable once `Ignite::releaseLockedTokens` is called; however, since the fee has already been paid and deducted from the user's stake amount, it will not be returned with the refunded `QI` stake. This behavior differs from the other registration methods, which all refund the usually non-refundable fee in the event of registration failure.

Impact: Users who register with a hosted Zeeve validator will not be refunded the Ignite fee if registration fails.

Recommended Mitigation: Refund the Ignite fee if registration fails for pre-validated `QI` stakes.

BENQI: Fixed in commit f671224.

Cyfrin: Verified. The fee is now taken from successful registrations during the call to `Ignite::releaseLockedTokens`.

7.3 Low Risk

7.3.1 StakingContract refunds are affected by global parameter updates

Description: When StakingContract::refundStakedAmount is called by the BENQI admin, the following validation is performed using the globally-defined refundPeriod:

```
require(
    block.timestamp > record.timestamp + refundPeriod,
    "Refund period not reached"
);
```

The StakingContract::StakeRecord struct does not have a corresponding member and so does not store the value of refundPeriod at the time of staking; however, if StakingContract::setRefundPeriod is called with an updated period then that of an existing record could be shorter/longer than expected.

Impact: The refund period for existing records could be affected by global parameter updates.

Recommended Mitigation: Consider adding an additional member to the StakeRecord struct to store the value of refundPeriod at the time of staking.

BENQI: Acknowledged, working as expected.

Cyfrin: Acknowledged.

7.3.2 Insufficient validation of Chainlink price feeds

Description: Validation of the price and updatedAt values returned by Chainlink AggregatorV3Interface::latestRoundData is performed within the following functions:

- StakingContract::_validateAndSetPriceFeed
- StakingContract::_getPriceInUSD
- Ignite::_initialisePriceFeeds
- Ignite::registerWithStake
- Ignite::registerWithErc20Fee
- Ignite::registerWithPrevalidatedQiStake
- Ignite::addPaymentToken
- Ignite::configurePriceFeed

However, there is additional validation shown below that is recommended but currently not present:

```
(uint80 roundId, int256 price, , uint256 updatedAt, ) = priceFeed.latestRoundData();
if(roundId == 0) revert InvalidRoundId();
if(updatedAt == 0 || updatedAt > block.timestamp) revert InvalidUpdate();
```

Impact: The impact is limited because the most important price and staleness validation are already present.

Recommended Mitigation: Consider including this additional validation and consolidating it into a single internal function.

BENQI: Acknowledged, current validation is deemed sufficient.

Cyfrin: Acknowledged.

7.3.3 Incorrect operator when validating subsidisation cap

Description: When a new registration is created, `Ignite::_registerWithChecks` validates that the subsidisation amount for the registration does not cause the maximum to be exceeded when added to the existing total subsidised amount:

```
require(
    totalSubsidisedAmount + subsidisationAmount < maximumSubsidisationAmount,
    "Subsidisation cap exceeded"
);
```

However, the incorrect operator is used when performing this comparison.

Impact: Registrations that cause the maximum subsidization amount to be met exactly will revert.

Recommended Mitigation:

```
    require(
-       totalSubsidisedAmount + subsidisationAmount < maximumSubsidisationAmount,
+       totalSubsidisedAmount + subsidisationAmount <= maximumSubsidisationAmount,
        "Subsidisation cap exceeded"
    );
```

BENQI: Fixed in commit 37446f6.

Cyfrin: Verified. The operator has been changed.

7.3.4 `StakingContract::slippage` can be outside of `minSlippage` or `maxSlippage`

Description: When the BENQI admin sets the `slippage` state by calling `StakingContract::setSlippage`, it is validated that this new value is between the `minSlippage` and `maxSlippage` thresholds:

```
require(
    _slippage >= minSlippage && _slippage <= maxSlippage,
    "Slippage must be between min and max"
);
```

The admin can also change both `minSlippage` and `maxSlippage` via `StakingContract::setMinSlippage` and `StakingContract::setMaxSlippage`; however, neither of these functions has any validation that the `slippage` state variable remains within the boundaries.

Impact: Incorrect usage of either `StakingContract::setMaxSlippage` or `StakingContract::setMinSlippage` can result in the `slippage` state variable being outside the range.

Proof of Concept: The following test can be added to `stakingContract.test.js` under `describe("setSlippage")`:

```
it("can have slippage outside of max and min", async function () {
    // slippage is set to 4
    await stakingContract.connect(benqiAdmin).setSlippage(4);
    // max slippage is updated below it
    await stakingContract.connect(benqiAdmin).setMaxSlippage(3);

    const slippage = await stakingContract.slippage();
    const maxSlippage = await stakingContract.maxSlippage();
    expect(slippage).to.be.greaterThan(maxSlippage);
});
```

Recommended Mitigation: Consider validating that the `slippage` state variable is within the boundaries set using `setMin/MaxSlippage()`.

BENQI: Fixed in commits 96e1b96 and dbb13c5.

Cyfrin: Verified.

7.3.5 Lack of user-defined slippage and deadline parameters in `StakingContract::swapForQI` may result in unfavorable `QI` token swaps

Description: When a user interacts with `StakingContract` to provision a hosted node, they can choose between two methods:`StakingContract::stakeWithAVAX` or `StakingContract::stakeWithERC20`. If the staked token is not `QI`, `StakingContract::swapForQI` is invoked to swap the staked token for `QI` via Trader Joe. Once created, the validator node is then registered with `Ignite`, using `QI`, via `StakingContract::registerNode`.

Within the swap to `QI`, `amountOutMin` is calculated using Chainlink price data and a slippage parameter defined by the protocol:

```
// Get the best price quote
uint256 slippageFactor = 100 - slippage; // Convert slippage percentage to factor
uint256 amountOutMin = (expectedQiAmount * slippageFactor) / 100; // Apply slippage
```

If the actual amount of `QI` received is below this `amountOutMin`, the transaction will revert; however, users are restricted by the protocol-defined slippage, which may not reflect their preferences if they desire a smaller slippage tolerance to ensure they receive a more favorable swap execution.

Additionally, the swap deadline specified as `block.timestamp` in `StakingContract::swapForQI` provides no protection as deadline validation will pass whenever the transaction is included in a block:

```
uint256 deadline = block.timestamp;
```

This could expose users to unfavorable price fluctuations and again offers no option for users to provide their own deadline parameter.

Impact: Users may receive fewer `QI` tokens than expected due to the fixed slippage tolerance set by the protocol, potentially resulting in unfavorable swap outcomes.

Recommended Mitigation: Consider allowing users to provide a `minAmountOut` slippage parameter and a `deadline` parameter for the swap operation. The user-specified `minAmountOut` should override the protocol's slippage-adjusted amount if larger.

BENQI: Acknowledged, there is already a slippage check inside `StakingContract::swapForQI` based on the Chainlink pricing.

Cyfrin: Acknowledged.

7.4 Informational

7.4.1 `AccessControlUpgradeable::_setupRole` is deprecated

Description: In `ValidatorRewarder::initialize` the `DEFAULT_ADMIN_ROLE` is assigned using `AccessControlUpgradeable::_setupRole`:

```
_setupRole(DEFAULT_ADMIN_ROLE, _admin);
```

This method has been deprecated by OpenZeppelin in favor of the `AccessControlUpgradeable::_grantRole` as written in their documentation and NatSpec:

```
/**
 * @dev Grants `role` to `account`.
 * ...
 * NOTE: This function is deprecated in favor of {_grantRole}.
 */
function _setupRole(bytes32 role, address account) internal virtual {
    _grantRole(role, account);
}
```

Note that `Ignite::initialize` also uses this, but since the contract is already initialized it is of no concern.

Recommended Mitigation: Consider using `AccessControlUpgradeable::_grantRole` in `ValidatorRewarder::initialize`, and possibly also in `Ignite::initialize`.

BENQI: Fixed in commit 8db7fb5.

Cyfrin: Verified.

7.4.2 Unchained initializers should be called instead

Description: While not an immediate issue in the current implementation, the direct use of initializer functions rather than their unchained equivalents should be avoided. `ValidatorRewarder::initialize` and `StakingContract::initialize` should be modified to avoid potential duplicate initialization in the future.

Note that this is also relevant for `Ignite::initialize`, but since the contract is already initialized it is of no concern.

Recommended Mitigation: Consider using unchained initializers in `ValidatorRewarder::initialize`, `StakingContract::initialize`, and possibly also in `Ignite::initialize`.

BENQI: Fixed in commit cd4d43e.

Cyfrin: Verified. Unchained initializers are now used.

7.4.3 Missing `onlyInitializing` modifier in `StakingContract`

Description: While it is not currently possible for the functions to be invoked elsewhere, both `StakingContract::initializeRoles` and `StakingContract::setInitialParameters` should be limited to being called during initialization but are missing the `onlyInitializing` modifier. Note that the latter is however handled by its internal call to `StakingContract::_initializePriceFeeds` that does have it applied.

Recommended Mitigation: Consider adding the `onlyInitializing` modifier to `StakingContract::initializeRoles` and possibly also `StakingContract::setInitialParameters`.

BENQI: Fixed in commit cd4d43e.

Cyfrin: Verified. The modifier has been added.

7.4.4 Unnecessary `amount` parameter in `StakingContract::stakeWithERC20`

Description: When provisioning a node through `StakingContract::stakeWithERC20`, users can pay with a supported ERC20 token. The `totalRequiredToken` is calculated based on the `avaxStakeAmount` (needed to register the node in Ignite) and the `hostingFee` (paid to Zeeve for hosting), before being transferred from the user to the contract:

```
require(isTokenAccepted(token), "Token not accepted");
uint256 hostingFee = calculateHostingFee(duration);

uint256 totalRequiredToken = convertAvaxToToken(
    token,
    avaxStakeAmount + hostingFee
);

require(amount >= totalRequiredToken, "Insufficient token");

// Transfer tokens from the user to the contract
IERC20Upgradeable(token).safeTransferFrom(
    msg.sender,
    address(this),
    totalRequiredToken
);
```

The `amount` parameter provided by the user is only used to validate that it covers the `totalRequiredToken`, but since execution will revert if the user has not given the contract sufficient allowance for the transfer, the `amount` parameter becomes redundant.

Recommended Mitigation: Consider removing the `amount` parameter from `StakingContract::stakeWithERC20`.

BENQI: Acknowledged. The purpose is to ensure that the value passed in by the frontend is not lower than the `totalRequiredToken`, acting as a form of slippage check. If it's lesser than totalRequiredToken, more tokens could be deducted than the user expected.

Cyfrin: Acknowledged.

7.4.5 Staking amount in QI should be calculated differently

Description: Currently, if the stake token is `QI`, `stakingAmountInQi` is calculated as shown below:

```
stakingAmountInQi = totalRequiredToken - convertAvaxToToken(token, hostingFee);
```

However, this can result in a precision loss of 1 wei.

Proof of Concept: This was tested using a Forge fixture and logs within the source code.

Recommended Mitigation: Consider calculating `stakingAmountInQi` directly based on `avaxStakeAmount`.

BENQI: Acknowledged. 1 wei precision loss is fine.

Cyfrin: Acknowledged.

7.4.6 Tokens with more than `18` decimals will not be supported

Description: Currently, tokens with more than `18` decimals are not supported due to the decimals handling logic in `StakingContract::_getPriceInUSD`:

```
uint256 decimalDelta = uint256(18) - tokenDecimalDelta;
```

and `Ignite::registerWithErc20Fee`:

```
uint tokenAmount = uint(avaxPrice) * registrationFee / uint(tokenPrice) / 10 ** (18 - token.decimals());
```

Recommended Mitigation: Modify this logic if tokens with a larger number of decimals are required to be supported.

BENQI: Acknowledged, working as expected.

Cyfrin: Acknowledged.

7.4.7 Incorrect revert strings in `StakingContract::revokeAdminRole`

Description: There are two revert strings [1, 2], shown below, in `StakingContract::revokeAdminRole` that appear to have been copied incorrectly and should respectively instead be:

- "Cannot revoke role from the zero address"
- "Attempting to revoke an unrecognized role"

```
function revokeAdminRole(bytes32 role, address account) public {
    // Ensure the account parameter is not a zero address to prevent accidental misassignments
    require(
        account != address(0),
        "Cannot assign role to the zero address"
    );

    /* snip: other conditionals */

    } else {
        // Optionally handle cases where an unknown role is attempted to be granted
        revert("Attempting to grant an unrecognized role");
    }

    /*snip: internal call & event emission */
}
```

Recommended Mitigation: Modify the revert strings as suggested.

BENQI: Fixed in commits cd4d43e and 99d7f25.

Cyfrin: Verified. The revert strings have been modified.

7.4.8 Placeholder recipient constants in `Ignite` should be updated before deployment

Description: While it is understood that the `FEE_RECIPIENT` and `SLASHED_TOKEN_RECIPIENT` constants in `Ignite` have been modified for testing purposes, it is important to note that they should be reverted to valid values before deployment to ensure that fees and slashed tokens are not lost.

```
address public constant FEE_RECIPIENT = 0xaAaAaAaaAaAaAaaAaAAAAAAAAaaaAaAaAaaAaaAa; // @audit-info -
↪  update placeholder values
address public constant SLASHED_TOKEN_RECIPIENT = 0xbBbBBBBbbBBBbbbBbbBbbbbBBbBbbbbBbBbbBBbB;
```

Recommended Mitigation: Update the constants before performing the `Ignite` contract upgrade.

BENQI: Acknowledged, already in the checklist for deployments.

Cyfrin: Acknowledged.

7.4.9 Missing modifiers

Description: Despite the use of OpenZeppelin libraries for re-entrancy guards and pausable functionality, not all external functions have the `nonReentrant` and pausable modifiers applied, so cross-function re-entrancy may be possible and functions could be called when not intended. Specifically:

- `Ignite::registerWithStake`, unlike other registration functions, is missing the `whenNotPaused` modifier.
- `Ignite::registerWithPrevalidatedQiStake` is missing both the `nonReentrant` and `whenNotPaused` modifiers.
- `StakingContract::registerNode`, which calls `Ignite::registerWithPrevalidatedQiStake`, does not have the `whenNotPaused` modifier applied either.
- The `whenPaused` and `whenNotPaused` modifiers are not applied to any of the pausable functions in both contracts. This is not strictly required but prescient to note.

Recommended Mitigation: Add the necessary modifiers where appropriate.

BENQI: The registration functions call `_register()` which enforces pause checks. The `nonReentrant` modifier was not added to `Ignite::registerWithPrevalidatedStake` since it is a permissioned function with no unsafe external calls. The `whenNotPaused` modifier has been added to `StakingContract::registerNode` in commit 4956824.

Cyfrin: Verified.

7.4.10 Incorrect assumption that Chainlink price feeds will always have the same decimals

Description: There are several instances [1, 2, 3] in `Ignite` where the decimal precision of Chainlink price feeds are assumed to be equal. Currently, this does not cause any issues as both `AVAX`, `QI`, and all other `USD` feeds return prices with `8` decimal precision, but this should be handled explicitly as `ETH` feeds return prices with `18` decimal precision as explained here.

Recommended Mitigation: Consider explicit handling of Chainlink price feed decimals.

BENQI: Acknowledged, USD feeds always have eight decimals.

Cyfrin: Acknowledged.

7.4.11 Typo in `Ignite::registerWithPrevalidatedQiStake` NatSpec

Description: There is a typo in the `Ignite::registerWithPrevalidatedQiStake` NatSpec.

Recommended Mitigation:

```diff
  /**
   * @notice Register a new node with a prevalidated QI deposit amount
-  * @param  beneficiary User no whose behalf the registration is made
+  * @param  beneficiary User on whose behalf the registration is made
   * @param  nodeId Node ID of the validator
   * @param  blsProofOfPossession BLS proof of possession (public key + signature)
   * @param  validationDuration Duration of the validation in seconds
   * @param  qiAmount The amount of QI that was staked
  */
```

BENQI: Fixed in commit 50c7c1a.

Cyfrin: Verified.

7.4.12 Magic numbers should be replaced by constant variables

Description: The magic numbers `10_000`, `2000e18`, `201/201e18` are used throughout the `Ignite` contract but should be made constant variables instead.

Recommended Mitigation: Use constants in place of the magic numbers outlined above.

BENQI: Acknowledged, won't change.

Cyfrin: Acknowledged.

7.4.13 Misalignment of `pause()` and `unpause()` access controls across contracts

Description: All three contracts, `Ignite`, `ValidatorRewarder`, and `StakingContract`, have pausing functionality that can be triggered by accounts with special privileges; however, they all implement the access control differently:

- In `Ignite`, `pause()` can only be called by accounts granted the `ROLE_PAUSE` role and similarly for `unpause()` it is the `ROLE_UNPAUSE` role.
- In `ValidatorRewarder`, both `pause()` and `unpause()` can only be called by accounts granted the `ROLE_PAUSE` role. The role `ROLE_UNPAUSE` is defined but not used.
- In `StakingContract`, both `pause()` and `unpause()` are limited to accounts granted the role `DEFAULT_ADMIN_ROLE`.

Recommended Mitigation: Consider aligning the role configuration between all contracts, preferably using the `ROLE_PAUSE`/`ROLE_UNPAUSE` setup from `Ignite` as it gives the most flexibility.

BENQI: Acknowledged, won't change.

Cyfrin: Acknowledged.

7.4.14 Inconsistent price validation in `Ignite::registerWithStake`

Description: In `Ignite::registerWithErc20Fee`, `Ignite::registerWithPrevalidatedQiStake`, and `StakingContract::_getPriceInUSD`, prices are validated to be greater than `0`; however, in `Ignite::registerWithStake`, the `AVAX` price is validated to be greater than the `QI` price. While the `AVAX` price is currently significantly higher than the `QI` price and so will not result in any unwanted reverts, this validation is inconsistent with the other instances and should be modified.

Recommended Mitigation: Modify the validation to require the `AVAX` price to be greater than `0` instead of the `QI` price.

BENQI: Acknowledged, won't change.

Cyfrin: Acknowledged.

7.5 Gas Optimization

7.5.1 Unnecessary validation of EnumerableSet functions

Description: When invoking EnumerableSet::add and EnumerableSet::remove, it is not necessary to first check whether an element already exists within the set as these functions perform the same validation internally. Instead, the return values should be checked.

Instances include: StakingContract::addToken, StakingContract::removeToken, Ignite::addPaymentToken, and Ignite::removePaymentToken.

Recommended Mitigation: Consider the following diff as an example:

```
function addToken(
    address token,
    address priceFeedAddress,
    uint256 maxPriceAge
) external onlyRole(BENQI_ADMIN_ROLE) {
-    require(!acceptedTokens.contains(token), "Token already exists");

    _validateAndSetPriceFeed(token, priceFeedAddress, maxPriceAge);
-    acceptedTokens.add(token);
+    require(acceptedTokens.add(token), "Token already exists");
    emit TokenAdded(token);
}
```

BENQI: Fixed in commits 4956824 and 420ace6.

Cyfrin: Verified. The validation has been updated.

7.5.2 Unnecessary validation in StakingContract::registerNode

Description: When a new validator node has been created on behalf of a user, the Zeeve admin reports this by calling StakingContract::registerNode which performs some validation before invoking Ignite::registerWithPrevalidatedQiStake to register the node according to the requirements in Ignite.

Some of this validation done in StakingContract::registerNode, shown below, is unnecessary and can be removed.

```
require(
    bytes(nodeId).length > 0 && blsProofOfPossession.length > 0,
    "Invalid node or BLS key"
);
require(
    igniteContract.registrationIndicesByNodeId(nodeId) == 0,
    "Node ID already registered"
);
```

All of this validation around nodeId, blsProofOfPossesion, and the registration index is performed again in Ignite::_register.

```
// Retrieve the staking details from the stored records
require(stakeRecords[user].stakeCount > 0, "Staking details not found");
require(index < stakeRecords[user].stakeCount, "Index out of bounds"); // Ensures the index is valid

StakeRecord storage record = stakeRecords[user].records[index]; // Access the record by index
```

If these requirements were removed, an invalid index or zero stake count would result in an uninitialized ‘StakeRecord being returned. Thus, execution would revert on all of the subsequent requirements:

```
require(record.timestamp != 0, "Staking details not found");
require(isValidDuration(record.duration), "Invalid duration");
// Ensure the staking status is Provisioning
require(
    record.status == StakingStatus.Provisioning,
    "Invalid staking status"
);
```

Even still, the timestamp validation is superfluous as there is no way for an existing record to have an uninitialized `timestamp`, and the record is guaranteed to exist by the subsequent check on `status`. This means that the `duration` validation is also unnecessary, as it is not needed to guarantee the existence of a record and is performed again in `Ignite::_regiserWithChecks`.

Recommended Mitigation: Consider removing the unnecessary validation outlined above.

BENQI: Acknowledged. Kept as a redundancy check.

Cyfrin: Acknowledged.

7.5.3 Unnecessary conditional block in `Ignite::getTotalRegistrations` can removed

Description: The conditional in `Ignite::getTotalRegistrations` is intended to handle the case where there are no registrations aside from the default placeholder registration; however, this is unnecessary because the function would still return `0` without this check if the `registrations.length` is `1`, and due to the presence of the default placeholder registration, it should not be possible to reach a state where `registrations.length` is `0`.

```
function getTotalRegistrations() external view returns (uint) {
    if (registrations.length <= 1) {
        return 0;
    }

    // Subtract 1 because the first registration is a dummy registration
    return registrations.length - 1;
}
```

Recommended Mitigation: Consider removing the conditional block.

BENQI: Fixed in commit 58af671.

Cyfrin: Verified. Validation has been removed.

7.5.4 Unnecessary validation in `Ignite::getRegistrationsByAccount`

Description: In `Ignite::getRegistrationsByAccount`, there is validation performed on the indices passed as arguments to the function:

```
require(from < to, "From value must be lower than to value");
require(to <= numRegistrations, "To value must be at most equal to the number of registrations");
```

This is not necessary as the call will revert due to underflow here or index out-of-bounds here. In the case `to == from`, an empty array would be returned.

Recommended Mitigation: Consider removing the validation shown above.

BENQI: Fixed in commit 82cf4fc.

Cyfrin: Verified. Validation has been removed.

7.5.5 Unnecessary price feed address validation in `Ignite::configurePriceFeed`

Description: Unlike `Ignite::addPaymentToken`, which performs no validation on the price feed address itself, and only the data it returns, `Ignite::configurePriceFeed` first validates that the price feed address is not `address(0)`. This is not necessary as the call to `AggregatorV3Interface::latestRoundData` would revert during abi decoding of the return data.

Proof of Concept: The following standalone Forge test can be used to demonstrate this:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.15;

import "forge-std/Test.sol";

contract TestZeroAddressCall is Test {
    function test_zeroAddressCall() external {
        vm.expectRevert(); // see: https://book.getfoundry.sh/cheatcodes/expect-revert#gotcha:~:text=%E⌋
        ↪  2%9A%A0%EF%B8%8F%20Gotcha%3A%20Usage%20with%20low%2Dlevel%20calls
        (bool revertsAsExpected, bytes memory returnData) =
            address(0).call(abi.encodeWithSelector(AggregatorV3Interface.latestRoundData.selector));
        assertFalse(revertsAsExpected); // the call itself does not revert

        vm.expectRevert(); // it's the decode step that reverts
        (uint80 roundId, int256 answer, uint256 startedAt, uint256 updatedAt, uint80 answeredInRound) =
            abi.decode(returnData, (uint80, int256, uint256, uint256, uint80));
    }
}

interface AggregatorV3Interface {
    function latestRoundData()
        external
        view
        returns (uint80 roundId, int256 answer, uint256 startedAt, uint256 updatedAt, uint80
        ↪  answeredInRound);
}
```

Recommended Mitigation: Consider removing the validation.

BENQI: Fixed in commit 7cbe588.

Cyfrin: Verified. Validation has been removed and the corresponding test has been updated.