# Dolomite - Proof of Liquidity (POL) Vaults Audit Report

Prepared by Cyfrin

Version 2.0


**Lead Auditors**

0kage

Farouk


April 24, 2025

# 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

The Dolomite Berachain Protocol implements a sophisticated system for managing yield-bearing assets while maintaining the security benefits of isolation mode in DeFi lending. It enables users to stake their assets in Berachain's Proof of Liquidity (POL) ecosystem while simultaneously using those assets as collateral in Dolomite's lending platform.

Key components of the Dolomite Berachain system include:

1. **Vault Architecture**: The protocol implements a dual-vault system:
   - *User Vaults*: Isolated vaults that represent a user's position within DolomiteMargin
   - *Meta Vaults*: Companion vaults that interact with Berachain's rewards systems, staking tokens to generate yield
2. **Isolation Mode Framework**: Each asset operates in isolation mode, preventing cross-contamination risk between different collateralized positions:
   - Enforces collateral type restrictions to limit systemic risk
   - Implements secure unwrapping and wrapping mechanisms between isolation mode tokens and underlying assets
3. **Rewards Integration:** The protocol connects with Berachain's rewards ecosystem:
   - *Infrared Integration*: Allows for staking in the Infrared protocol
   - *BGT Support*: Handles Berachain Governance Token (BGT) and other token rewards
   - *Reward Harvesting*: Automatically claims and routes rewards to appropriate user vaults
4. **Trader System**: Implements specialized traders for moving assets between systems:
   - *Wrapper Traders*: Convert underlying tokens to isolation mode tokens
   - *Unwrapper Traders*: Convert isolation mode tokens back to underlying tokens

- *Generic Trader Proxy*: Coordinates multi-step trades across different markets

5. **Registry System**: Central registry contracts maintain addresses and relationships:

The protocol solves a key challenge in DeFi by allowing users to earn yield on their collateral assets while maintaining the safety of isolated risk positions. This "stake-and-borrow" architecture enables capital efficiency while promoting participation in Berachain's governance and rewards systems.

# 5 Audit Scope

The Dolomite Protocol implements an advanced ecosystem for handling Proof of Liquidity (POL) assets on Berachain, enabling users to stake and use their assets as collateral while participating in the Berachain rewards ecosystem. The protocol leverages isolation mode mechanics to maintain proper risk isolation while allowing assets to earn rewards through underlying vaults.

The audit focused on identifying potential security vulnerabilities, logical errors, and adherence to best practices within the smart contracts. Special attention was given to the isolation mode mechanics, staking/unstaking process, asset transfer flows, and the interactions between vaults, meta-vaults, and the DolomiteMargin protocol.

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

**Core Protocol**

- GenericTraderProxyBase.sol
- GenericTraderProxyV2.sol
- GenericTraderProxyV2Lib.sol
- BerachainRewardsRegistry.sol
- IsolationModeTokenVaultV1.sol
- SimpleIsolationModeVaultFactory.sol
- IsolationModeVaultFactory.sol

**POL (Proof of Liquidity) Components**

- POLIsolationModeTokenVaultV1.sol
- POLIsolationModeVaultFactory.sol
- POLIsolationModeTraderBaseV2.sol
- POLIsolationModeWrapperTraderV2.sol
- POLIsolationModeUnwrapperTraderV2.sol
- POLIsolationModeWrapperUpgradeableProxy.sol
- POLIsolationModeUnwrapperUpgradeableProxy.sol
- POLPriceOracleV2.sol

**Infrared Vault Components**

- InfraredBGTIsolationModeTokenVaultV1.sol
- InfraredBGTIsolationModeVaultFactory.sol
- InfraredBGTMetaVault.sol
- MetaVaultRewardReceiver.sol
- MetaVaultRewardTokenFactory.sol
- MetaVaultUpgradeableProxy.sol

# 6 Executive Summary

Over the course of 16 days, the Cyfrin team conducted an audit on the Dolomite - Proof of Liquidity (POL) Vaults smart contracts provided by Dolomite Protocol. In this period, a total of 9 issues were found.

The Dolomite Berachain Protocol implements a sophisticated system for managing yield-bearing assets within an isolation mode framework, enabling users to participate in Berachain's Proof of Liquidity (POL) ecosystem while using those assets as collateral within Dolomite's lending platform.

The security audit focused on the core vault architecture (User Vaults and Meta Vaults), the isolation mode mechanics, reward integration systems, trading mechanisms between different asset types, and liquidation processes. The review encompassed all key components including vault contracts, trader implementations, the rewards registry, and proxy patterns implemented throughout the system.

The audit identified a generally well-engineered codebase with strong security practices, although several issues require resolution before production deployment. Most notably, a high-risk vulnerability allows permissionless reward claims that could permanently lock funds in a metavault. Additionally, a medium-risk issue was discovered related to restaking iBGT rewards during infrared staking exits. **All major issues were successfully mitigated by the Dolomite team.**

The Dolomite Berachain protocol demonstrates strong security practices including:

- Exceptional test coverage with comprehensive unit and integration tests for various scenarios
- Clear separation of concerns between vault management, trading, and rewards handling
- Well-implemented proxy patterns for upgradability across critical components
- Robust vault architecture that maintains isolation while enabling yield generation
- Secure trading mechanisms for moving between isolation mode assets and underlying tokens

The test suite is particularly impressive, covering nearly all edge cases and complex interactions between different components of the system. While the workflows were generally well laid out in visual flowcharts, the in-line comments and overall documentation could be improved given the complexity of the Dolomite ecosystem and its integration with Berachain's reward mechanisms. Additional high-level documentation explaining the relationships between components would benefit future developers and auditors.

**Summary**

| | |
|---|---|
| Project Name | Dolomite - Proof of Liquidity (POL) Vaults |
| Repository | dolomite-margin-modules |
| Commit | 9e428aa2712c. . . |
| Audit Timeline | Mar 20th - April 10th |
| Methods | Manual Review, Stateful Fuzzing |

**Issues Found**

| | |
|---|---|
| Critical Risk | 0 |
| High Risk | 1 |
| Medium Risk | 1 |
| Low Risk | 3 |
| Informational | 3 |
| Gas Optimizations | 1 |
| Total Issues | 9 |

**Summary of Findings**

| | |
|---|---|
| [H-1] InfraVault's Permissionless Reward Claiming Can Allow Anyone to Lock Rewards in the MetaVault | Resolved |
| [M-1] User rewards may remain unclaimed after calling `InfraredBGTIsolationModeTokenVaultV1::exit()` function | Resolved |
| [L-1] Missing validation for initialization calldata in `POLIsolationModeWrapperUpgradeableProxy` constructor | Acknowledged |
| [L-2] Reward loss risk when transitioning between reward vault types | Acknowledged |
| [L-3] Rewards in iBGT cannot be redeemed when infrared vault staking is paused | Resolved |
| [I-1] Redundant check for non-zero reward amount in `_handleRewards` function | Acknowledged |
| [I-2] Inconsistent ETH handling pattern in Proxy Contracts | Resolved |
| [I-3] Missing event emissions for some important state changes | Resolved |
| [G-1] `_handleRewards` gas optimisation | Acknowledged |

# 7 Findings

## 7.1 High Risk

### 7.1.1 InfraVault's Permissionless Reward Claiming Can Allow Anyone to Lock Rewards in the MetaVault

**Description:** The `InfraredBGTMetaVault` relies on `_performDepositRewardByRewardType` to handle newly claimed rewards by either depositing them into DolomiteMargin or sending them directly to the vault owner. However, the onchain Infrared vault contract allows anyone to call `getRewardForUser` on any user's rewards. Note that this function is defined in the `MultiRewards.sol`, the contract that infrared vault derives from.

This can trigger a reward transfer to the MetaVault unexpectedly. Because the code that deposits or forwards these tokens (`_performDepositRewardByRewardType`) only runs during the normal "self-claim" flow, rewards triggered through a third-party call would not go through the intended deposit or distribution logic.

```
/// @inheritdoc IMultiRewards
function getRewardForUser(address _user)
    public
    nonReentrant
    updateReward(_user)
{
    onReward();
    uint256 len = rewardTokens.length;
    for (uint256 i; i < len; i++) {
        address _rewardsToken = rewardTokens[i];
        uint256 reward = rewards[_user][_rewardsToken];
        if (reward > 0) {
            (bool success, bytes memory data) = _rewardsToken.call{
                gas: 200000
            }(
                abi.encodeWithSelector(
                    ERC20.transfer.selector, _user, reward
                )
            );
            if (success && (data.length == 0 || abi.decode(data, (bool)))) {
                rewards[_user][_rewardsToken] = 0;
                emit RewardPaid(_user, _rewardsToken, reward);
            } else {
                continue;
            }
        }
    }
}
```

**Impact:** An attacker could force rewards to be sent to the MetaVault's address without triggering `_performDepositRewardByRewardType`. As a result, those newly arrived tokens could stay in the MetaVault contract, never being staked or deposited into DolomiteMargin or distributed to the vault owner.

We note that the token loss is not permanent as the InfraredBGTMetaVault contract is upgradeable. Nevertheless this can cause delays as every user has an independent vault and upgrading each vault would be cumbersome. In the meanwhile, vault owners cannot use their received rewards within the Dolomite Protocol.

**Proof of Concept:** Consider the following scenario: 1. An attacker calls `infravault.getRewardForUser(metaVaultAddress)`. 2. The reward is transferred to metaVaultAddress rather than going through the `_performDepositRewardByRewardType` logic. 3. The tokens remain stuck in the MetaVault contract if there is no fallback mechanism to move or stake them again.

**Recommended Mitigation:** Consider modifying the `_performDepositRewardByRewardType` to add the token balance in the vault to the reward amount and routing all BGT token vault staking into Infrared vaults via the metavault.

```
  function _performDepositRewardByRewardType(
        IMetaVaultRewardTokenFactory _factory,
```

```
        IBerachainRewardsRegistry.RewardVaultType _type,
        address _token,
        uint256 _amount
    ) internal {
++ _amount += IERC20(token).balanceOf(address(this));
}
```

**Dolomite:** Fixed in d0a638a.

**Cyfrin:** Verified

## 7.2 Medium Risk

### 7.2.1 User rewards may remain unclaimed after calling `InfraredBGTIsolationModeTokenVaultV1::exit()` function

**Description:** When a user calls `InfraredBGTIsolationModeTokenVaultV1::exit` in a scenario where rewards are in the same token (iBGT), the function correctly unstakes their original deposit but:

- The rewards are credited to the user's Dolomite Margin account balance
- The same tokens are simultaneously re-staked in the Infrared vault

This creates a situation where users may believe they've fully exited the protocol, but in reality, they have rewards still staked. The `_exit()` function calls `_handleRewards()` which processes any rewards earned. The issue occurs when `_handleRewards()` automatically deposits iBGT rewards back into Dolomite Margin and re-stakes them:

```
function _exit() internal {
    IInfraredVault vault = registry().iBgtStakingVault();

    IInfraredVault.UserReward[] memory rewards = vault.getAllRewardsForUser(address(this));
    vault.exit();

    _handleRewards(rewards);
}

function _handleRewards(IInfraredVault.UserReward[] memory _rewards) internal {
    IIsolationModeVaultFactory factory = IIsolationModeVaultFactory(VAULT_FACTORY());
    for (uint256 i = 0; i < _rewards.length; ++i) {
        if (_rewards[i].amount > 0) {
            if (_rewards[i].token == UNDERLYING_TOKEN()) {
                _setIsDepositSourceThisVault(true);
                factory.depositIntoDolomiteMargin(
                    DEFAULT_ACCOUNT_NUMBER,
                    _rewards[i].amount
                ); //@audit restakes the reward amount
                assert(!isDepositSourceThisVault());
            } else {
                // Handle other tokens...
            }
        }
    }
}
```

**Impact:**

- Users may never realize they still have assets staked in the protocol
- Rewards remain locked in a separate vault that users might not know to check
- For smaller reward amounts, gas costs might exceed the value, effectively trapping these funds
- Poor user experience when "exit" doesn't fully exit the protocol

**Proof of Concept:** Add the following test to the `#exit` class of tests in `InfraredBGTIsolationModeTokenVaultV1.ts`

```
    it('should demonstrate that iBGT rewards are re-staked after exit', async () => {
      await testInfraredVault.setRewardTokens([core.tokens.iBgt.address]);
      await core.tokens.iBgt.connect(iBgtWhale).approve(testInfraredVault.address, rewardAmount);
      await testInfraredVault.connect(iBgtWhale).addReward(core.tokens.iBgt.address, rewardAmount);
      await registry.connect(core.governance).ownerSetIBgtStakingVault(testInfraredVault.address);

      await iBgtVault.depositIntoVaultForDolomiteMargin(defaultAccountNumber, amountWei);
      await expectProtocolBalance(core, iBgtVault, defaultAccountNumber, iBgtMarketId, amountWei);
```

```
    // Get   the initial staking balance before exit
    const initialStakingBalance = await testInfraredVault.balanceOf(iBgtVault.address);
    expect(initialStakingBalance).to.eq(amountWei);

    // Call exit which should unstake original amount but rewards will be re-staked
    await iBgtVault.exit();

    // Check that staking balance equals reward amount (rewards were re-staked)
    const finalStakingBalance = await testInfraredVault.balanceOf(iBgtVault.address);
    expect(finalStakingBalance).to.eq(rewardAmount, "Staking balance should equal reward amount after
    ↪  exit");

    // Verify original deposit is in the wallet (but not rewards)
    await expectWalletBalance(iBgtVault, core.tokens.iBgt, amountWei);

    // Verify protocol balance now includes both original deposit and rewards
    await expectProtocolBalance(core, iBgtVault, defaultAccountNumber, iBgtMarketId,
    ↪  amountWei.add(rewardAmount));
  });
```

**Recommended Mitigation:** Consider avoiding restaking when the user calls `exit` explicitly - it is counter-intuitive even from a UX standpoint to restake assets in the same vault when the user has expressed intent to exit the vault completely. Also add clear documentation explaining how rewards are handled across different vaults

**Dolomite:** Fixed in d0a638a.

**Cyfrin:** Verified.

## 7.3 Low Risk

### 7.3.1 Missing validation for initialization calldata in `POLIsolationModeWrapperUpgradeableProxy` constructor

**Description:** The `POLIsolationModeWrapperUpgradeableProxy` constructor accepts initialization calldata without performing any validation on its content before executing it via `delegatecall` to the implementation contract. Specifically:

- The constructor does not verify that the `calldata` targets the expected initialize(address) function
- The constructor does not verify that the provided vaultFactory address parameter is non-zero

```
// POLIsolationModeWrapperUpgradeableProxy.sol
constructor(
    address _berachainRewardsRegistry,
    bytes memory _initializationCalldata
) {
    BERACHAIN_REWARDS_REGISTRY = IBerachainRewardsRegistry(_berachainRewardsRegistry);
    Address.functionDelegateCall(
        implementation(),
        _initializationCalldata,
        "POLIsolationModeWrapperProxy: Initialization failed"
    );
}
```

This lack of validation means the constructor will blindly execute any calldata, potentially setting critical contract parameters incorrectly during deployment.

Note that a similar issue exists in `POLIsolationModeUnwrapperUpgradeableProxy`

**Impact:** The proxy could be initialized with a zero or invalid vaultFactory address, rendering it non-functional or insecure. Additionally, if the implementation contract is upgraded and introduces new functions with weaker access controls, this pattern would allow those functions to be called during initialization of new proxies.

**Recommended Mitigation:** Consider adding explicit validation for both the function selector and parameters in the constructor:

```
constructor(
    address _berachainRewardsRegistry,
    bytes memory _initializationCalldata
) {
    BERACHAIN_REWARDS_REGISTRY = IBerachainRewardsRegistry(_berachainRewardsRegistry);

    // Validate function selector is initialize(address)
    require(
        _initializationCalldata.length == 36 &&
        bytes4(_initializationCalldata[0:4]) == bytes4(keccak256("initialize(address)")),
        "Invalid initialization function"
    );

     // Decode and validate the vaultFactory address is non-zero
    address vaultFactory = abi.decode(_initializationCalldata[4:], (address));
    require(vaultFactory != address(0), "Zero vault factory address");

    Address.functionDelegateCall(
        implementation(),
        _initializationCalldata,
        "POLIsolationModeWrapperProxy: Initialization failed"
    );
}
```

**Dolomite:** Acknowledged.

**Cyfrin:** Acknowledged.

### 7.3.2 Reward loss risk when transitioning between reward vault types

**Description:** When a user's default reward vault type for an asset is changed (e.g., from NATIVE or BGTM to INFRARED) in the `InfraredBGTMetaVault` contract, current logic attempts to claim outstanding rewards before switching the type. However, the implementation fails to retrieve rewards from the user's current vault type, leading to permanent loss of accrued rewards.

The issue occurs in the relationship between `_setDefaultRewardVaultTypeByAsset` and `_getReward` functions:

- When staking tokens via `_stake`, the function calls `_setDefaultRewardVaultTypeByAsset` to ensure the asset uses the INFRARED reward type.
- If the asset's current reward type is not INFRARED, `_setDefaultRewardVaultTypeByAsset` calls `_getReward` to claim pending rewards for the asset before changing the type.
- However, `_getReward` hardcodes the reward vault type to INFRARED instead of using the user's current reward type

```
function _getReward(address _asset) internal {
    IBerachainRewardsRegistry.RewardVaultType rewardVaultType =
    ↪  IBerachainRewardsRegistry.RewardVaultType.INFRARED;
    IInfraredVault rewardVault = IInfraredVault(REGISTRY().rewardVault(
        _asset,
        rewardVaultType  // Always uses INFRARED, ignoring user's current type
    ));
    // ... claim rewards logic ...
}
```

This means that when transitioning from another reward type (e.g., NATIVE or BGTM) to INFRARED, the contract attempts to claim rewards from the INFRARED vault even though the user's rewards are accrued in a different vault type.

Additionally, the assertion that should prevent changing types when a user has a staked balance is ineffective for non-INFRARED types:

```
assert(getStakedBalanceByAssetAndType(_asset, currentType) == 0);
```

This assertion will always pass for non-INFRARED types because `getStakedBalanceByAssetAndType` only can have non-zero balances for INFRARED due to the `onlyInfraredType` modifier on all staking functions.

**Impact:** While the protocol currently only intends to support the INFRARED reward type, this issue creates a potential risk for future expansion.

If/when the protocol adds support for additional reward types (NATIVE, BGTM) and users accrue rewards in these vaults, they would permanently lose these rewards when transitioning to the INFRARED type. Once the registry is updated to use INFRARED as the default reward type, the rewards in the original vault become inaccessible through normal contract interactions.

The severity of this issue depends on the protocol's roadmap for supporting multiple reward types. If there are definite plans to expand beyond INFRARED, this represents a significant risk of permanent reward loss for users.

**Proof of Concept:** Consider following scenario:

- In a future version, assume a user has accrued rewards in a NATIVE reward vault
- User calls a function that triggers `_setDefaultRewardVaultTypeByAsset` to transition to INFRARED
- The assertion `assert(getStakedBalanceByAssetAndType(_asset, currentType) == 0)` passes because balances in this contract are only tracked for INFRARED
- `_getReward(_asset)` is called but retrieves from INFRARED vault instead of NATIVE vault

- The registry is updated via `REGISTRY().setDefaultRewardVaultTypeFromMetaVaultByAsset(_asset, _type)`
- User's rewards in the NATIVE vault are now inaccessible

**Recommended Mitigation:** If the protocol plans to support multiple reward types in the future, modify the `_getReward` function to claim rewards from the user's current reward vault type.

Additionally, considering implementing proper balance tracking for all reward types if multiple types will be supported, or clearly document that transitioning between reward types requires manual reward claiming first.

If only INFRARED vault is supported, consider removing `_setDefaultRewardVaultTypeByAsset` as it is not serving any purpose. Since this is called in the `stake` function, removing this will simplify the code and save gas.

**Dolomite:** Acknowledged. We know code will have to change a good bit to allow multiple reward types.

**Cyfrin** Acknowledged.

### 7.3.3 Rewards in iBGT cannot be redeemed when infrared vault staking is paused

**Description:** Current reward handling mechanism enforces automatic reinvestment of iBGT rewards back into the staking pool, without providing an alternative when staking is disabled.

When the Infrared protocol pauses staking (which can happen for various reasons such as security emergencies or technical issues), users are left with no way to access their earned rewards, effectively freezing these assets until staking is resumed. It is noteworthy that InfraredVault does not prevent redeeming rewards/ unstaking even when staking is paused

The issue lies in the `_handleRewards` function, which automatically attempts to reinvest iBGT rewards:

```
function _handleRewards(IInfraredVault.UserReward[] memory _rewards) internal {
    IIsolationModeVaultFactory factory = IIsolationModeVaultFactory(VAULT_FACTORY());
    for (uint256 i = 0; i < _rewards.length; ++i) {
        if (_rewards[i].amount > 0) {
            if (_rewards[i].token == UNDERLYING_TOKEN()) {
                _setIsDepositSourceThisVault(true);
                factory.depositIntoDolomiteMargin(
                    DEFAULT_ACCOUNT_NUMBER,
                    _rewards[i].amount
                );
                assert(!isDepositSourceThisVault());
            } else {
                // ... handle other token types ...
            }
        }
    }
}
```

When the staking function in the Infrared vault is paused, as it can be through the pauseStaking() function...

```
/// @inheritdoc IInfraredVault
function pauseStaking() external onlyInfrared {
    if (paused()) return;
    _pause();
}
```

...the staking operation in `executeDepositIntoVault` will fail due to the whenNotPaused modifier in the InfraredVault:

```
function stake(uint256 amount) external whenNotPaused {
     // code
```

**Impact:** Users on Dolomite are unable to access their earned rewards during periods when staking is paused in the Infrared vault even though such redemption is allowed on Infrared vaults.

**Proof of Concept:** The `TestInfraredVault` is made `Pausable` to align with the on-chain Infrared vault contract and `whenNotPaused` modifier is added to the `stake` function.

```
contract TestInfraredVault is ERC20, Pausable {

   function unpauseStaking() external {
        if (!paused()) return;
        _unpause();
    }

    function pauseStaking() external {
        if (paused()) return;
        _pause();
    }

    function stake(uint256 amount) external whenNotPaused {
        _mint(msg.sender, amount);
        IERC20(asset).transferFrom(msg.sender, address(this), amount);
    }
}
```

Add the following test to the #getReward class of tests in `InfraredBGTIsolationModeTokenVaultV1.ts`

```
  it('should revert when staking is paused and rewards are in iBGT', async () => {
      await testInfraredVault.setRewardTokens([core.tokens.iBgt.address]);

      // Add iBGT as reward token and fund the reward
      await core.tokens.iBgt.connect(iBgtWhale).approve(testInfraredVault.address, rewardAmount);
      await testInfraredVault.connect(iBgtWhale).addReward(core.tokens.iBgt.address, rewardAmount);
      await registry.connect(core.governance).ownerSetIBgtStakingVault(testInfraredVault.address);

      // Deposit iBGT into the vault
      await iBgtVault.depositIntoVaultForDolomiteMargin(defaultAccountNumber, amountWei);
      await expectProtocolBalance(core, iBgtVault, defaultAccountNumber, iBgtMarketId, amountWei);

      // Advance time to accumulate rewards
      await increase(ONE_DAY_SECONDS * 30);

      // Pause staking in the InfraredVault
      await testInfraredVault.pauseStaking();
      expect(await testInfraredVault.paused()).to.be.true;

      //Calling getReward should revert because reinvesting iBGT rewards will fail due to staking being
      ↪  paused
      await expectThrow(
        iBgtVault.getReward()
      );


      // Verify balances remain unchanged
      await expectWalletBalance(iBgtVault, core.tokens.iBgt, ZERO_BI);
      await expectProtocolBalance(core, iBgtVault, defaultAccountNumber, iBgtMarketId, amountWei);

      // Unpause to allow normal operation to continue
      await testInfraredVault.unpauseStaking();
      expect(await testInfraredVault.paused()).to.be.false;

      // Now getReward should succeed
      await iBgtVault.getReward();
```

```
    await expectWalletBalance(iBgtVault, core.tokens.iBgt, ZERO_BI);
    await expectProtocolBalance(core, iBgtVault, defaultAccountNumber, iBgtMarketId,
    ↪  amountWei.add(rewardAmount));

    // Verify the reward was restaked
    expect(await testInfraredVault.balanceOf(iBgtVault.address)).to.eq(amountWei.add(rewardAmount));
  });
```

**Recommended Mitigation:** Consider checking if the BGT staking vault is `paused` before attempting to re-invest the rewards. If vault is paused, rewards can either be retained in the vault or transferred back to the vault owner.

**Dolomite:** Fixed in 7b83e77.

**Cyfrin:** Verified

## 7.4 Informational

### 7.4.1 Redundant check for non-zero reward amount in `_handleRewards` function

**Description:** In `InfraredBGTIsolationModeTokenVaultV1.sol`, the `_handleRewards` function includes a redundant check for positive reward amounts, as shown below:

```
//InfraredBGTIsolationModeTokenVaultV1.sol
function _handleRewards(IInfraredVault.UserReward[] memory _rewards) internal {
    IIsolationModeVaultFactory factory = IIsolationModeVaultFactory(VAULT_FACTORY());
    for (uint256 i = 0; i < _rewards.length; ++i) {
        if (_rewards[i].amount > 0) {  // <-- This check is redundant
            if (_rewards[i].token == UNDERLYING_TOKEN()) {
                _setIsDepositSourceThisVault(true);
                factory.depositIntoDolomiteMargin(
                    DEFAULT_ACCOUNT_NUMBER,
                    _rewards[i].amount
                );
                assert(!isDepositSourceThisVault());
            } else {
                // ... rest of function
            }
        }
    }
}
```

This check is redundant because the getAllRewardsForUser function in the InfraredVault only returns rewards with a positive amount:

```
// InfraredVault.sol
function getAllRewardsForUser(address _user) external view returns (UserReward[] memory) {
    uint256 len = rewardTokens.length;
    UserReward[] memory tempRewards = new UserReward[](len);
    uint256 count = 0;
    for (uint256 i = 0; i < len; i++) {
        uint256 amount = earned(_user, rewardTokens[i]);
        if (amount > 0) {  // @audit <-- Already filtering for positive amounts
            tempRewards[count] = UserReward({token: rewardTokens[i], amount: amount});
            count++;
        }
    }
    // Create a new array with the exact size of non-zero rewards
    UserReward[] memory userRewards = new UserReward[](count);
    for (uint256 j = 0; j < count; j++) {
        userRewards[j] = tempRewards[j];
    }
    return userRewards;
}
```

**Recommended Mitigation:** Consider removing the redundant check.

**Dolomite:** No longer applicable. Code changed a good bit because of bricked rewards fix

**Cyfrin** Acknowledged.

### 7.4.2 Inconsistent ETH handling pattern in Proxy Contracts

**Description:** There is an inconsistency in how proxy contracts handle incoming ETH transactions through their `receive()` and `fallback()` functions. Some proxy contracts delegate both functions to their implementation, while others only delegate the fallback() function while leaving the receive() function empty.

For example, in `MetaVaultUpgradeableProxy.sol`, both functions delegate:

```
// MetaVaultUpgradeableProxy
receive() external payable requireIsInitialized {
    _callImplementation(implementation());
}

fallback() external payable requireIsInitialized {
    _callImplementation(implementation());
}
```

Whereas in `POLIsolationModeWrapperUpgradeableProxy.sol` and `POLIsolationModeUnwrapperUpgradeableProxy`, only the `fallback()` function delegates:

```
// POLIsolationModeWrapperUpgradeableProxy
receive() external payable {} // solhint-disable-line no-empty-blocks

fallback() external payable {
    _callImplementation(implementation());
}
```

While this is a design choice and not a security issue per se, it could lead to potential confusion among developers who might expect all proxies to handle ETH transfers in a similar manner.

**Recommended Mitigation:** Consider documenting the chosen approach and reasoning in the contract comments to clarify the intended behavior for other developers.

**Dolomite:** Fixed d4ceeef.

**Cyfrin:** Verified.

#### 7.4.3 Missing event emissions for some important state changes

**Description:** The POL contracts are missing event emissions for several important state-changing operations.

1. *POLIsolationModeTokenVaultV1*
   - `prepareForLiquidation`: No event when a position is prepared for liquidation
   - `stake/unstake`: No event for (un)staking actions
   - `getReward`: No event for reward claims
   - `exit`: No event for exiting positions
2. *InfraredBGTMetaVault*
   - `chargeDTokenFee`: No event for fee charging
3. *POLIsolationModeTraderBaseV2*
   - `_POLIsolationModeTraderBaseV2__initialize`: No event when vault factory is set
4. *MetaVaultRewardTokenFactory*
   - `depositIntoDolomiteMarginFromMetaVault`: No event for deposit from meta vault
   - `depositIntoDolomiteMarginFromMetaVault`: No event for deposit of other token from meta vault

**Recommended Mitigation:** Consider reviewing the codebase and adding events that track important state changes.

**Dolomite:** Fixed in e556252 and ccfcd12.

**Cyfrin:** Verified.

## 7.5 Gas Optimization

### 7.5.1 `_handleRewards` gas optimisation

**Description:** `InfraredBGTMetaVault._performDepositRewardByRewardType` is a function that will be called every time rewards are fetched from Infrared vault. The function can be optimized as follows:

- Remove the reward amount > 0 check (as listed in *Redundant check for non-zero reward amount in `_handleRewards` function* )
- Cache frequently used functions `DOLOMITE_MARGIN()`, `OWNER()`
- Cache reward token and reward amount at the start of the loop
- Use unchecked integer for incrementing reward counter

**Recommended Mitigation:** Consider using the below optimized version:

```
function _handleRewards(IInfraredVault.UserReward[] memory _rewards) internal {
    IIsolationModeVaultFactory factory = IIsolationModeVaultFactory(VAULT_FACTORY());
    address owner = OWNER();
    IDolomiteMargin dolomiteMargin = DOLOMITE_MARGIN();

    for (uint256 i = 0; i < _rewards.length;) {
        //@audit Removed redundant check since InfraredVault only sends non-zero rewards
        address rewardToken = _rewards[i].token;
        uint256 rewardAmount = _rewards[i].amount;

        if (rewardToken == UNDERLYING_TOKEN()) {
            _setIsDepositSourceThisVault(true);
            factory.depositIntoDolomiteMargin(
                DEFAULT_ACCOUNT_NUMBER,
                rewardAmount
            );
            assert(!isDepositSourceThisVault());
        } else {
        try dolomiteMargin.getMarketIdByTokenAddress(rewardToken) returns (uint256 marketId) {
                        IERC20(rewardToken).safeApprove(address(dolomiteMargin), rewardAmount);
                        try factory.depositOtherTokenIntoDolomiteMarginForVaultOwner(
                            DEFAULT_ACCOUNT_NUMBER,
                            marketId,
                           rewardAmount
                        ) {} catch {
                            IERC20(rewardToken).safeApprove(address(dolomiteMargin), 0);
                            IERC20(rewardToken).safeTransfer(owner, rewardAmount);
                        }
                    } catch {
                        IERC20(rewardToken).safeTransfer(owner,  rewardAmount);
                    }
        }
        unchecked { ++i; }
    }
}
```

**Dolomite:** No longer applicable. Code changed a good bit because of bricked rewards fix.

**Cyfrin:** Acknowledged.