D2 finance Audit Report

Prepared by Cyfrin

Version 2.1

Lead Auditors

Immeas

February 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 D2 contracts audited function similarly to a traditional ETF. A user deposits assets into a Vault, where the funds are managed by the protocol and actively traded or invested using the D2 strategy over the course of an epoch. The strategy integrates with various DeFi protocols across multiple chains: The integrations include:

- **On Berachain:** Kodiak, Ooga Booga, Infrared, and Dolomite.
- **On Arbitrum One, Base, and Ethereum:** 1Inch, Aave, Pendle, GMX V2, Dolomite, and Silo.

Once the epoch ends, the trader returns the funds, and the proceeds are distributed among the vault shareholders in proportion to their holdings.

4.1 Actors and Roles

- **1. Actors:**
  - **Project Admin:** Deploys vaults and strategies, configures settings, and manages epoch start and end times.
  - **Trader:** Executes trades using the D2 strategies.
  - **Users:** Vault shareholders who deposit funds and withdraw their share of the trading proceeds.
- **2. Roles:**
  - `ADMIN_ROLE`**:**
    - Deploys and configures epoch lengths.
    - Manages whitelisted users for the vaults and the assets/amounts users can hold to bypass the whitelist.
    - Can emergency start/end an epoch unless the contract is frozen.
    - Can freeze the ability to emergency start/end an epoch.

    - Can change the assigned vault in the Trader module.
    - Custodies and returns funds in the Trader module.
    - Can set withdrawal fees, adjust fee rates, and change the fee recipient in the Trader module.
    - Can add facets to the Strategy unless the functionality is frozen.
    - Can freeze the contract to prevent any new facets from being added.
  - `EXECUTOR_ROLE`:
    - Conducts all trading activities across different modules.
    - Defines swap paths and slippage parameters for trades.

4.2 Key Components

- `VaultV0`: A less transparent, ad-hoc trading vault. Here, funds are not custodied by the `Strategy` contract but instead by a protocol-controlled wallet. This allows trading on protocols that do not yet have written integrations.
- `VaultV3`: The primary vault, which interacts with the `Strategy` diamond contract. Users deposit assets, which are then custodied by the `Strategy`, where the trader employs D2 strategies.
- `Strategy`: A diamond-style contract with immutable facets, each representing an integration with an external protocol.

4.3 VaultV3 Flow

1. The project admin creates a vault, setting the funding period and epoch length.
2. Users join by depositing assets during the funding period.
3. The epoch starts, and the trader custodies the funds from the vault.
4. The trader utilizes different modules in the Strategy to trade, invest, and earn yield for users.
5. The epoch ends, and the trader exits all positions and returns funds to the vault.
6. Users can withdraw their share of the earned profits.

4.4 Centralization Risks

As outlined above, the D2 contracts rely on two centralized roles. The `EXECUTOR_ROLE`, a potocol controlled wallet, is of particular importance as it is responsible for active trading. This reliance introduces potential risks, including the possibility of private key leaks.

To mitigate these risks, we encourage the protocol to adopt a more decentralized and defensive smart contract design. Implementing strict slippage parameter for swaps and ensuring that the value going into an operation closely matches the value coming out can enhance security and reduce reliance on centralized control.

5 Audit Scope

Cyfrin conducted an audit of D2 based on the code present in the repository commit hash `c2fc257`. Cyfrin also audited the additions of a Kodiak Island integration in commit `ca2e36e` as well as admin emergency calls added in commit `ab5b852`

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

- `contracts/modules/Aave.sol`
- `contracts/modules/Bera.sol` - Inludes integrations to Bera staking, Kodiak, Ooga Booga, Infrared and Dolomite

- `contracts/modules/Camelot.sol`
- `contracts/modules/D2.sol`
- `contracts/modules/Dolomite.sol`
- `contracts/modules/GMXV2.sol` - GMXV2 had the `GMXV2_Module` contract commented out, which we treated as in-scope and uncommented
- `contracts/modules/IDolomite.sol`
- `contracts/modules/Inch.sol`
- `contracts/modules/Pendle.sol`
- `contracts/modules/Silo.sol`
- `contracts/modules/Trader.sol`
- `contracts/modules/WETH.sol`
- `contracts/D2OFT.sol`
- `contracts/Strategy.sol`
- `contracts/VaultV0.sol`
- `contracts/VaultV3.sol`

6 Executive Summary

Over the course of 26 days, the Cyfrin team conducted an audit on the D2 finance smart contracts provided by D2. In this period, a total of 24 issues were found.

The review identified two critical issues:

1. A GMX V2 integration issue where the strategy address was passed as a callback contract when entering a position, causing withdrawals to fail. The Strategy contract did not implement the necessary functions, meaning that any funds in GMX V2 would be permanently lost.
2. A deployer account's private key was found in clear text within one of the project files. Although this file was not part of the audit scope, storing private keys in plaintext is a serious security risk and an anti-pattern that must be strictly avoided.

The review also uncovered two high-severity issues:

1. An issue with the combination of ERC-4626 vaults and LayerZero's cross-chain functionality for vault share tokens. Transferring tokens to another chain would effectively surrender the user's share of the vault's assets.
2. A misconfigured interface definition prevented rewards from being claimed from Berachain's RewardVault contracts.

Additionally, the audit discovered multiple medium- and low-risk issues. While these are less severe, we strongly recommend addressing them to improve the protocol's security posture.

The audited contracts lacked test coverage, so the Cyfrin team developed a Foundry test suite as part of the audit. This included:

- Unit tests for VaultV0, VaultV3, the core Strategy contract, and the Trader module.
- Fork tests for all external protocol integrations, which uncovered integration issues (as noted above).
- An Echidna fuzz test suite for VaultV3.

All tests and findings were delivered to the protocol as a PR upon audit completion.

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	D2 finance
Repository	d2-contracts
Commit	c2fc257605eb...
Audit Timeline	Jan 10th - Feb 14th
Methods	Manual Review, Stateful Fuzzing

Issues Found

Critical Risk	2
High Risk	2
Medium Risk	7
Low Risk	7
Informational	6
Gas Optimizations	0
Total Issues	24

Summary of Findings

[C-1] `GMXV2_Module` withdrawal functionality is broken due to missing callback implementation	Resolved
[C-2] `PRIVATE_KEY` exposed in `makefile` file and other sensitive data	Resolved
[H-1] Users may lose value when transferring ERC-4626 Vault tokens cross-chain	Resolved
[H-2] Reward claiming fails for Berachain RewardVaults due to incorrect interface	Resolved
[M-1] Lack of slippage protection allows exploitation of Pendle trades	Resolved
[M-2] Unsafe token transfers in `VaultV3` can lead to state inconsistencies	Resolved
[M-3] Incorrect `chainid` prevents correct Strategy deployment on Berachain	Resolved
[M-4] Incorrect interface for `ExchangeRouter::updateOrder` prevents order updates in GMX V2	Resolved

[M-5] `Bera_Module` fails to stake in `BGTStaker` due to missing Boost integration	Resolved
[M-6] `Bera_Module::bera_kodiakv3_swap` broken due to `deadline` parameter	Resolved
[M-7] Kodiak swap functions do not check if output token is approved, risking stuck assets	Resolved
[L-1] Aave's module lacks reward-claiming functionality	Resolved
[L-2] Compromised trader account can block admin from revoking access	Resolved
[L-3] Improper deadline handling	Acknowledged
[L-4] Missing events for important state changes	Acknowledged
[L-5] `Silo_Module::silo_execute` will revert as approval is to the wrong contract	Resolved
[L-6] Silo and Dolomite lack LTV limit increasing liquidation risk	Acknowledged
[L-7] Borrowing non-approved tokens can bypass trading restrictions	Resolved
[I-1] `ETH` calls on `Dolomite_Module` have no corresponding calls on Dolomite	Resolved
[I-2] Unnecessary validation in `Bera_Module::bera_infrared_stake` should follow standard pattern	Resolved
[I-3] Unused imports	Resolved
[I-4] Aave `swapBorrowRateMode` is deprecated	Resolved
[I-5] Selectors for `Bera_Module::bera_kodiakv2_swap` and `bera_kodiakv3_swap` needs to be separately added	Resolved
[I-6] Consider using Ownable2Step instead of Ownable	Acknowledged

7 Findings

7.1 Critical Risk

7.1.1 `GMXV2_Module` withdrawal functionality is broken due to missing callback implementation

Description: The `GMXV2_Module` contract's withdrawal functionality is broken because it sets itself (`address(this)`) as the callback contract for GMX V2 withdrawals without implementing the required callback interface. This causes all withdrawal transactions to revert, effectively locking user funds in the protocol.

```
function gmxv2_withdraw(
    address market,
    uint256 executionFee,
    uint256 amount,
    uint256 minLongOut,
    uint256 minShortOut
) external payable onlyRole(EXECUTOR_ROLE) nonReentrant {
    IExchangeRouter.CreateWithdrawalParams memory params = IExchangeRouter.CreateWithdrawalParams({
        receiver: address(this),
        callbackContract: address(this),  // @audit - Setting callback without implementation
        // ...
    });
    // ...
}
```

According to GMX V2's documentation and source code, when a callback contract is specified, it must implement two critical functions:

1. `afterWithdrawalExecution(bytes32, WithdrawalUtils.Props, EventUtils.EventLogData)`
2. `afterWithdrawalCancellation(bytes32, WithdrawalUtils.Props)`

These callbacks are enforced by GMX's `ExecuteWithdrawalUtils.sol`:

```
// In ExecuteWithdrawalUtils.sol
function executeWithdrawal(...) {
    // ... other code ...
    CallbackUtils.afterWithdrawalExecution(params.key, withdrawal, eventData);
    // ... other code ...
}

// In CallbackUtils.sol
function afterWithdrawalExecution(...) internal {
    if (withdrawal.callbackContract() != address(0)) {
        IWithdrawalCallbackReceiver(withdrawal.callbackContract()).afterWithdrawalExecution(
            key,
            withdrawal,
            eventData
        );
    }
}
```

Impact: This issue completely breaks a core protocol function. It has severe implications:

- **Complete loss of functionality**: All withdrawal transactions through `gmxv2_withdraw` will revert, making it impossible for users to withdraw their funds.
- **Loss of funds**: User funds become locked in the protocol.

Proof of Concept: We've created a test that demonstrates this issue:

```
function test_gmxv2_withdraw_RevertGivenMissingCallback() public {
        // First make a deposit to get market tokens
        uint256 depositAmount = 1e18;
```

```
        deal(WETH_ADDRESS, address(strategy), depositAmount);
        vm.prank(address(strategy));
        _weth.approve(GMX_DEPOSIT_VAULT, depositAmount);

        // Store initial balance
        uint256 initialExecutorBalance = EXECUTOR.balance;

        // Get market info before deposit
        IMarket.Props memory marketInfo = IReader(GMX_READER).getMarket(GMX_DATASTORE, GMX_WETH_MARKET);
        address marketToken = marketInfo.marketToken;

        // Execute deposit
        vm.prank(EXECUTOR);
        _gmx.gmxv2_deposit{value: 0.1 ether}(
            GMX_WETH_MARKET,
            0.1 ether,  // execution fee
            depositAmount,
            0,          // no short token
            0           // min out
        );

        // Deal market tokens to the vault first before simulating keeper execution
        deal(marketToken, GMX_DEPOSIT_VAULT, depositAmount);

        // ============ KEEPER SIMULATION - DEPOSIT EXECUTION ============
        // In production: GMX keeper would execute this step
        // Here we simulate the keeper minting market tokens to the strategy
        vm.prank(GMX_DEPOSIT_VAULT);
        IERC20(marketToken).transfer(address(strategy), depositAmount); // Simulate market tokens being
        ↪  minted
        // ============ END KEEPER SIMULATION - DEPOSIT EXECUTION ============

        // Get market token balance after deposit
        uint256 marketTokenBalance = IERC20(marketToken).balanceOf(address(strategy));

        require(marketTokenBalance > 0, "No market tokens received from deposit");

        // Approve market tokens for withdrawal
        vm.prank(address(strategy));
        IERC20(marketToken).approve(GMX_WITHDRAW_VAULT, depositAmount);

        // Now attempt to withdraw
        vm.prank(EXECUTOR);
        _gmx.gmxv2_withdraw{value: 0.1 ether}(
            GMX_WETH_MARKET,
            0.1 ether,  // execution fee
            depositAmount,
            0,          // min long out
            0           // min short out
        );

        // Deal WETH to the withdraw vault before simulating keeper execution
        deal(WETH_ADDRESS, GMX_WITHDRAW_VAULT, depositAmount * 2); // Ensure vault has enough WETH

        // First give market tokens to the strategy
        deal(marketToken, address(strategy), depositAmount);

        // ============ KEEPER SIMULATION - WITHDRAWAL EXECUTION ============
        // In production: GMX keeper would:
        // 1. Burn market tokens
        // 2. Return WETH
        // 3. Call the callback contract (if set) with afterWithdrawalExecution
```

```
        // Here we simulate steps 1 and 2 manually:
        vm.startPrank(GMX_WITHDRAW_VAULT);
        // Step 1: Burn market tokens (transfer them to withdraw vault)
        IERC20(marketToken).transferFrom(address(strategy), GMX_WITHDRAW_VAULT, depositAmount);
        // Step 2: Return WETH to strategy
        IERC20(WETH_ADDRESS).transfer(address(strategy), depositAmount);
        vm.stopPrank();
        // Note: Step 3 (callback) is not simulated as we're bypassing GMX's actual execution
        // ============ END KEEPER SIMULATION - WITHDRAWAL EXECUTION ============


        // Verify executor spent the execution fees
        assertEq(
            EXECUTOR.balance,
            initialExecutorBalance - 0.2 ether, // 0.1 for deposit + 0.1 for withdrawal
            "Executor should have spent 0.2 ETH for execution fees"
        );
    }
```

The test passes, but this is misleading because we're manually simulating the keeper's actions. In production, according to GMX V2's documentation, withdrawals require specific callback handling:

> "Ensure only handlers with the CONTROLLER role can call the afterWithdrawalExecution and afterWithdrawalCancellation callback functions"

This indicates that GMX V2 expects contracts interacting with withdrawals to implement these callback functions. When our contract attempts a withdrawal in production:

1. The keeper will attempt to call `afterWithdrawalExecution` on our contract
2. Since we don't implement this interface, the transaction will revert
3. The withdrawal will fail, leaving user funds stuck in the protocol

Our test passes only because we're bypassing GMX's actual withdrawal execution by manually simulating the token transfers, which skips the mandatory callback mechanism.

Recommended Mitigation: There are two possible fixes:

Option 1: Remove Callback (Recommended) If no post-withdrawal processing is needed:

```
IExchangeRouter.CreateWithdrawalParams memory params = IExchangeRouter.CreateWithdrawalParams({
    receiver: address(this),
    callbackContract: address(0),  // Set to zero address to skip callbacks
    // ...
});
```

Option 2: Implement Callback Interface If post-withdrawal processing is required, implement the IWithdrawalCallbackReceiver interface:

```
contract GMXV2_Module is IGMXV2_Module, IWithdrawalCallbackReceiver {
    function afterWithdrawalExecution(
        bytes32 key_,
        WithdrawalUtils.Props memory withdrawal_,
        EventUtils.EventLogData memory eventData_
    ) external {
        // Add post-withdrawal logic
    }

    function afterWithdrawalCancellation(
        bytes32 key_,
        WithdrawalUtils.Props memory withdrawal_
    ) external {
        // Add cancellation logic
```

```
    }
}
```

We recommend Option 1 (removing the callback) unless there's a specific need for post-withdrawal processing, as it's simpler and requires less gas.

D2: Fixed in `37df474`

Cyfrin: Verified.

7.1.2 `PRIVATE_KEY` exposed in `makefile` file and other sensitive data

Description: The project's Makefile contains hardcoded sensitive credentials including:

- A private key used for contract deployments
- Multiple RPC endpoint URLs with API keys
- Several Etherscan API keys for different networks

Impact: Although no customer funds were at risk and the deployer key had no admin rights on any of the deployed strategies, this exposure is critical because:

1. The private key grants complete control over the associated address, including:
    - Full access to any funds held by the wallet
    - Impersonate the protocol and deploy fake vaults
2. The RPC endpoints could be used to:
    - Execute unauthorized API calls
    - Potentially incur costs to the project
    - Exceed rate limits affecting production services

Recommended Mitigation: Immediate Actions Required:

- Remove all exposed credentials immediately
- Transfer any assets from the compromised address
- Revoke any contract permissions from the address
- Regenerate all API keys

Future mitigation use methods in this Updraft lesson to safely store private keys.

D2: Fixed in `a4f3517` and `228d0e17`

Cyfrin: Verified.

7.2 High Risk

7.2.1 Users may lose value when transferring ERC-4626 Vault tokens cross-chain

Description: Both the `VaultV0` and `VaultV3` ERC-4626 tokens are designed to be transferable across chains and therefore implement the LayerZero cross-chain OFT standard.

When a token is transferred cross-chain, the `_debit` function is called to determine how the token should be "removed" from the source chain. The two standard approaches for this are **Lock**/**Unlock** and **Mint**/**Burn**. Both `VaultV0` and `VaultV3` implement the **Mint**/**Burn** approach, as seen in `VaultV0::_debit` and `VaultV0::_credit` (with an identical implementation in `VaultV3`):

```
function _debit(address _from, uint256 _amountLD, uint256 _minAmountLD, uint32 _dstEid) internal
↪   virtual override returns (uint256 amountSentLD, uint256 amountReceivedLD) {
    (amountSentLD, amountReceivedLD) = _debitView(_amountLD, _minAmountLD, _dstEid);
    _burn(_from, amountSentLD);
}

function _credit(address _to, uint256 _amountLD, uint32) internal virtual override returns (uint256
↪   amountReceivedLD) {
    if (_to == address(0x0)) _to = address(0xdead);
    _mint(_to, _amountLD);
    return _amountLD;
}
```

However, this approach is problematic for ERC-4626 vaults because `_burn` reduces the `totalSupply`. Since share value is calculated as `assets / totalSupply`, transferring tokens to another chain increases the share value of stakers who still have their tokens on the vault's original chain. This means that if these stakers withdraw, they will receive a portion of the assets that originally belonged to users who transferred their tokens cross-chain.

Impact: Cross-chain transfers distort the vault's share value. Users who transfer their tokens risk losing part or all of their value if other users withdraw while their tokens are still in transit.

Proof of Concept: The following test shows the issue, a similar test was written for `VaultV3`:

```
function test_vaultV0_xchain_transfer_affects_vault_exchange_rate() public {
    deal(address(asset), DEPOSITOR, 2e18);

    // 1. Bob and Alice both participate in the vault
    vm.startPrank(DEPOSITOR);
    asset.approve(address(vault), 2e18);
    vault.deposit(1e18, ALICE);
    vault.deposit(1e18, BOB);
    assertEq(vault.balanceOf(ALICE), 1e18);
    assertEq(vault.balanceOf(BOB), 1e18);
    vm.stopPrank();

    // 2. Share price is not 1 to 1
    assertEq(vault.previewRedeem(1e18), 1e18);

    SendParam memory sendParam;
    sendParam.to = bytes32(uint256(1));
    sendParam.amountLD = 1e18;

    MessagingFee memory fee;

    // 3. Bob transfers his share to another chain
    vm.prank(BOB);
    vault.send(sendParam, fee, BOB);

    // Since the share is burnt on the source chain, share price has increased
    assertEq(vault.previewRedeem(1e18), 2e18);
```

```
    // 4. Alice redeems her share, getting Bobs share as well
    vm.prank(ALICE);
    vault.redeem(1e18,ALICE, ALICE);

    assertEq(vault.balanceOf(ALICE), 0);
    assertEq(asset.balanceOf(ALICE), 2e18);

    Origin memory origin;
    origin.sender = bytes32(uint256(1));
    (bytes memory message,) = OFTMsgCodec.encode(
        bytes32(uint256(uint160(BOB))),
        1e6,
        ""
    );

    // 5. Bob transfers back
    vm.prank(LZ_ENDPOINT);
    vault.lzReceive(
        origin,
        bytes32(uint256(0)),
        message,
        LZ_ENDPOINT,
        ""
    );

    // Bobs share is now worth nothing as Alice has redeemed it
    assertEq(vault.balanceOf(BOB), 1e18);
    assertEq(vault.previewRedeem(1e18), 0);

    vm.prank(BOB);
    vm.expectRevert("ZERO_ASSETS");
    vault.redeem(1e18, BOB, BOB);
}
```

Recommended Mitigation: Consider using the **Lock/Unlock** approach and custody the funds in the vault when transferred:

```
function _debit(address _from, uint256 _amountLD, uint256 _minAmountLD, uint32 _dstEid) internal
↪   virtual override returns (uint256 amountSentLD, uint256 amountReceivedLD) {
    (amountSentLD, amountReceivedLD) = _debitView(_amountLD, _minAmountLD, _dstEid);
    // Instead of burning, transfer to this contract
    _transfer(_from, address(this), amountSentLD);
}

function _credit(address _to, uint256 _amountLD, uint32) internal virtual override returns (uint256
↪   amountReceivedLD) {
    if (_to == address(0x0)) _to = address(0xdead);
    _transfer(address(this),_to, _amountLD);
    return _amountLD;
}
```

D2: Fixed in 21f2dd1 and 902a14b

Cyfrin: Verified.

7.2.2 Reward claiming fails for Berachain RewardVaults due to incorrect interface

Description: D2 plans to deploy on the newly launched Berachain, which operates on a novel Proof-of-Liquidity model. On Berachain, liquidity providers can stake their LP tokens in Reward Vaults to earn $BGT, the Berachain Governance Token.

To facilitate staking and withdrawals, the D2 Bera_Module includes the functions Bera_Module::bera_vault_stake

and Bera_Module::bera_vault_withdraw. To claim $BGT rewards, the Bera_Module::bera_vault_get_reward function is used:

```
function bera_vault_get_reward(address token) external onlyRole(EXECUTOR_ROLE) nonReentrant {
    IVault(vaultFactory.getVault(token)).getReward(address(this));
}
```

However, the interface used for IVault::getReward is incorrect:

```
function getReward(address account) external;
```

In Berachain's RewardVault::getReward and as deployed on-chain, the correct function signature is:

```
function getReward(
    address account,
    address recipient
)
```

Due to this discrepancy, any call to Bera_Module::bera_vault_get_reward will revert, preventing $BGT rewards from being claimed.

Impact: The $BGT rewards, which are a core incentive of Berachain's Proof-of-Liquidity model, cannot be claimed. This reduces the protocol's ability to effectively participate in Berachain's reward system. Additionally, since these rewards contribute to the protocol's overall earnings.

Proof of Concept: The following test highlights the issue:

```
function test_bera_vault_get_reward() public {
    deal(USDCe_HONEY_STABLE_ADDRESS, address(strategy), 1e18);

    vm.startPrank(EXECUTOR);
    trader.approve(USDCe_HONEY_STABLE_ADDRESS, USDCe_HONEY_STABLE_VAULT, 1e18);
    bera.bera_vault_stake(USDCe_HONEY_STABLE_ADDRESS, 1e18);

    assertEq(RewardVault(USDCe_HONEY_STABLE_VAULT).balanceOf(address(strategy)), 1e18);

    vm.warp(block.timestamp + 1 days);

    // there are rewards to be claimed
    assertGt(RewardVault(USDCe_HONEY_STABLE_VAULT).earned(address(strategy)), 0);

    // rewards cannot be claimed due to the interface definition being wrong
    vm.expectRevert();
    bera.bera_vault_get_reward(USDCe_HONEY_STABLE_ADDRESS);

    vm.stopPrank();
}
```

Recommended Mitigation: Consider adding the extra parameter to the interface and call it with address(this):

```
-   IVault(vaultFactory.getVault(token)).getReward(address(this));
+   IVault(vaultFactory.getVault(token)).getReward(address(this), address(this));
```

D2: Fixed in 92303a6

Cyfrin: Verified.

7.3 Medium Risk

7.3.1 Lack of slippage protection allows exploitation of Pendle trades

Description: The `Pendle_Module` is designed to interact with the Pendle protocol. Pendle is a permissionless yield-trading protocol that enables users to execute various yield-management strategies. It utilizes a complex system of different token types—`PT`, `YT`, and `SY`—to facilitate trading across different components of yield.

Within Pendle, users can deposit assets to receive these tokens and subsequently swap between them. The `Pendle_Module` facet implements these functionalities.

The issue, however, is that slippage protection is completely absent, as the slippage parameters are hardcoded to `0` in multiple functions:

- `Pendle_Module::pendle_deposit`, L69:

```
router.addLiquiditySingleToken(
    address(this),
    address(market),
    0, // @audit `minLpOut` hardcoded to 0
    approxParams,
    input,
    limitOrderData
);
```

- `Pendle_Module::pendle_withdraw`, L86:

```
IPRouter.TokenOutput memory output = IPRouter.TokenOutput({
    tokenOut: ast,
    minTokenOut: 0, // @audit `minTokenOut` hardcoded to 0
    tokenRedeemSy: ast,
    pendleSwap: address(0),
    swapData: swapData
});
```

- `Pendle_Module::pendle_swap`, L140, L150 and L170:

```
IPRouter.TokenOutput memory output = IPRouter.TokenOutput({
    tokenOut: ast,
    minTokenOut: 0, // @audit `minTokenOut` hardcoded to 0
    tokenRedeemSy: ast,
    pendleSwap: address(0),
    swapData: swapData
});
```

```
router.swapExactTokenForPt(
    address(this),
    market,
    0, // @audit `minPtOut` hardcoded to 0
    approxParams,
    input,
    limitOrderData
);
```

```
router.swapExactTokenForYt(
    address(this),
    market,
    0, // @audit `minYtOut` hardcoded to 0
    approxParams,
    input,
    limitOrderData
);
```

- Pendle_Module::pendle_claim, L197:

```
IPSY(sy).redeem(
    address(this),
    IERC20(sy).balanceOf(address(this)),
    IPSY(sy).getTokensOut()[0],
    0, // @audit `minTokenOut` is hardcoded to 0
    false
);
```

- Pendle_Module::pendle_exit, L216:

```
router.exitPostExpToToken(
    address(this),
    market,
    amount,
    amountLp,
    IPRouter.TokenOutput({
        tokenOut: asset,
        minTokenOut: 0, // @audit `minTokenOut` hardcoded to 0
        tokenRedeemSy: asset,
        pendleSwap: address(0),
        swapData: IPSwapAggregator.SwapData({
            swapType: IPSwapAggregator.SwapType.NONE,
            extRouter: address(0),
            extCalldata: "",
            needScale: false
        })
    })
);
```

Since all these functions allow transactions with zero slippage protection, they may be exploited by MEV bots causing value loss for the stakers in D2.

Impact: The absence of slippage protection exposes the protocol to front-running and price manipulation. Attackers can take advantage of these hardcoded zero values to extract value from trades at the expense of D2's stakers, leading to potential financial losses.

Recommended Mitigation: Consider allowing the trader to provide `minAmountOut` in the calls.

D2: Fixed in cd7058d

Cyfrin: Verified.

7.3.2 Unsafe token transfers in `VaultV3` can lead to state inconsistencies

Description: The `VaultV3` contract doesn't validate the success of token transfers in both `custodyFunds()` and `returnFunds()` functions. State changes occur before or regardless of transfer success, which can lead to vault state inconsistencies.

```
function custodyFunds() external onlyTrader notCustodied duringEpoch returns (uint256) {
    uint256 amount = totalAssets();
    require(amount > 0, "!amount");
    custodied = true;
    custodiedAmount = amount;
    IERC20(asset()).transfer(trader, amount); // No success validation
    emit FundsCustodied(epochId, amount);
    return amount;
}

function returnFunds(uint256 _amount) external onlyTrader {
    require(custodied, "!custody");
    require(_amount > 0, "!amount");
```

```
    IERC20(asset()).transferFrom(trader, address(this), _amount); // No success validation
    epoch.epochEnd = uint80(block.timestamp);
    custodied = false;
    emit FundsReturned(currentEpoch, _amount);
}
```

Impact: Even with a trusted trader, several non-malicious scenarios can trigger this issue:

1. Token transfers being temporarily paused (e.g., USDC during SVB crisis)
2. Missing or insufficient allowances for transferFrom
3. Token blacklisting or transfer limits
4. Token implementations that return false instead of reverting

When triggered, this can lead to:

- Vault becoming stuck in custodied state
- Incorrect epoch transitions
- Mismatched accounting of assets
- Temporary locking of user funds
- Need for manual governance intervention

Recommended Mitigation: Use OpenZeppelin's SafeERC20 library and follow the Checks-Effects-Interactions pattern:

```
import "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol";

contract VaultV3 {
    using SafeERC20 for IERC20;

    function custodyFunds() external onlyTrader notCustodied duringEpoch returns (uint256) {
        uint256 amount = totalAssets();
        require(amount > 0, "!amount");
        custodied = true;
        custodiedAmount = amount;
        IERC20(asset()).safeTransfer(trader, amount);
        emit FundsCustodied(epochId, amount);
        return amount;
    }

    function returnFunds(uint256 _amount) external onlyTrader {
        require(custodied, "!custody");
        require(_amount > 0, "!amount");
        epoch.epochEnd = uint80(block.timestamp);
        custodied = false;
        IERC20(asset()).safeTransferFrom(trader, address(this), _amount);
        emit FundsReturned(currentEpoch, _amount);
    }
}
```

D2: Fixed in e54eb5b

Cyfrin: Verified.

7.3.3 Incorrect `chainid` prevents correct Strategy deployment on Berachain

Description: The Strategy contract includes a special configuration specifying which facets should be used for different chains. However, the `chainid` assigned to Berachain is incorrect, as seen in `Strategy::constructor`, L216:

```
} else if (block.chainid == 80000) { // @audit Berachain id is 80094
```

According to the official documentation, the correct `chainid` for Berachain is `80094`, not `80000`.

Impact: Facets intended for deployment on Berachain will not be correctly initialized until a new Strategy contract is deployed with the corrected `chainid`. This prevents the expected functionality from being executed on Berachain.

Recommended Mitigation: Consider changing `chainid` to `80094`:

```
- } else if (block.chainid == 80000) {
+ } else if (block.chainid == 80094) {
```

D2: Fixed in commit ab5b852

Cyfrin: Verified.

7.3.4 Incorrect interface for `ExchangeRouter::updateOrder` prevents order updates in GMX V2

Description: There is an issue in the integration with the GMX V2 trading platform, specifically in the `GMX_Module::gmxv2_update` function:

```
function gmxv2_update(
    bytes32 key,
    uint256 sizeDeltaUsd,
    uint256 acceptablePrice,
    uint256 triggerPrice,
    uint256 minOutputAmount
) external onlyRole(EXECUTOR_ROLE) nonReentrant {
    exchangeRouter.updateOrder(key, sizeDeltaUsd, acceptablePrice, triggerPrice, minOutputAmount);
}
```

The issue arises from an incorrect interface definition for `ExchangeRouter::updateOrder`:

```
function updateOrder(
    bytes32 key,
    uint256 sizeDeltaUsd,
    uint256 acceptablePrice,
    uint256 triggerPrice,
    uint256 minOutputAmount
) external payable;
```

However, the actual function signature used by GMX in its `ExchangeRouter` is:

```
function updateOrder(
    bytes32 key,
    uint256 sizeDeltaUsd,
    uint256 acceptablePrice,
    uint256 triggerPrice,
    uint256 minOutputAmount,
    uint256 validFromTime,
    bool autoCancel
) external payable nonReentrant {
```

The discrepancy between the expected and actual function signatures means that calls to `GMX_Module::gmxv2_update` will always revert due to missing parameters.

Impact: The `gmxv2_update` function is non-functional and will always revert, preventing order updates. While this issue can be circumvented by closing and reopening positions, this workaround is inefficient and leads to unnecessary transaction costs.

Proof of Concept: The following test highlights the issue:

```
function test_gmxv2_update_SucceedGivenValidOrder() public {
    // First create a long position
    uint256 collateralAmount = 1e18;
    uint256 sizeDeltaUsd = 10000e30;
    uint256 initialTriggerPrice = 2500e30;  // Current ETH price
    uint256 initialAcceptablePrice = 2400e30;  // Lower than trigger price for market long (willing to
    ↪  pay up to this price)
    uint256 initialMinOutputAmount = 0.9e18;
    uint256 executionFee = 0.1 ether;

    // Deal WETH to the strategy
    deal(WETH_ADDRESS, address(strategy), collateralAmount);

    // Create the order
    vm.prank(address(strategy));
    _weth.approve(GMX_ORDER_VAULT, collateralAmount);

    vm.recordLogs();
    // Create a market increase order
    vm.prank(EXECUTOR);
    _gmx.gmxv2_create{value: executionFee}(
        GMX_WETH_MARKET,
        WETH_ADDRESS,
        new address[](0), // No swap path
        IExchangeRouter.OrderType.LimitIncrease,
        IExchangeRouter.DecreasePositionSwapType.NoSwap,
        true, // isLong
        sizeDeltaUsd,
        collateralAmount,
        initialTriggerPrice,
        initialAcceptablePrice,
        initialMinOutputAmount,
        executionFee
    );
    Vm.Log[] memory entries = vm.getRecordedLogs();

    // Get the order key
    bytes32 key = entries[5].topics[2];

    // Wait for a few blocks to ensure order is ready
    vm.warp(block.timestamp + 1 hours);
    vm.roll(block.number + 100);

    // New parameters for the update
    uint256 newSizeDeltaUsd = 20000e30;
    uint256 newAcceptablePrice = 2200e30;
    uint256 newTriggerPrice = 2050e30;
    uint256 newMinOutputAmount = 0.95e18;

    // Record initial executor balance
    uint256 initialBalance = EXECUTOR.balance;

    // Update the order
    // This will fail because the interface used is not the correct one
    vm.prank(EXECUTOR);
    _gmx.gmxv2_update(
```

```
        key,
        newSizeDeltaUsd,
        newAcceptablePrice,
        newTriggerPrice,
        newMinOutputAmount
    );

    // Verify the executor spent the correct amount of ETH for execution fees
    assertEq(
        EXECUTOR.balance,
        initialBalance,
        "Executor balance should not change for updates"
    );
}
```

Recommended Mitigation: Consider changing the interface and adding the extra parameters to the call.

D2: Fixed in 23a48bc

Cyfrin: Verified.

7.3.5 `Bera_Module` fails to stake in `BGTStaker` due to missing Boost integration

Description: The D2 Berachain integration includes two functions, `Bera_Module::bera_bgt_stake` and `Bera_Module::bera_bgt_withdraw`. These functions attempt to interact with `BGTStaker::stake` and `BGTStaker::withdraw`, respectively.

However, both `BGTStaker::stake` and `BGTStaker::withdraw` can only be called by the `BGT` token contract itself, as enforced by the `onlyBGT` modifier:

```
/// @dev Throws if called by any account other than BGT contract.
modifier onlyBGT() {
    if (msg.sender != address(stakeToken)) NotBGT.selector.revertWith();
    _;
}
...

function stake(address account, uint256 amount) external onlyBGT {
    _stake(account, amount);
}
```

To stake in `BGTStaker`, a user must use the **boost functionality** on the `BGT` token. This involves:

1. Calling `BGT::queueBoost` and `BGT::activateBoost` to boost a verifier.
2. Calling `BGT::queueDropBoost` and `BGT::dropBoost` to exit the boost.

More details about boosting verifiers can be found in the Berachain documentation.

Impact: The protocol will not be able to earn `$BGT` rewards from `BGTStaker` as intended. To access these rewards, it must integrate with the boost functionality of the `BGT` token instead of calling `BGTStaker` directly. This oversight could result in missed staking incentives, reducing the protocol's overall earnings.

Recommended Mitigation: Consider implementing the boost functionality on the BGT token

D2: Fixed in 11c9407

Cyfrin: Verified.

7.3.6 `Bera_Module::bera_kodiakv3_swap` broken due to `deadline` parameter

Description: The function `Bera_Module::bera_kodiakv3_swap` calls Kodiak's `SwapRouter02::exactInput` (which is similar to Uniswap V3) to execute a swap:

```
function bera_kodiakv3_swap(address token, uint amount, uint amountMin, bytes calldata path) external
↪  onlyRole(EXECUTOR_ROLE) nonReentrant {
    validateToken(token);
    IERC20(token).approve(address(kodiakv3swap), amount);
    kodiakv3swap.exactInput(IKodiakV3.ExactInputParams({
        path: path,
        recipient: address(this),
        deadline: type(uint256).max,
        amountIn: amount,
        amountOutMinimum: amountMin
    }));
}
```

However, the `deadline` parameter is included in the function call but is not present in the SwapRouter02.ExactInputParams struct:

```
struct ExactInputParams {
    bytes path;
    address recipient;
    uint256 amountIn;
    uint256 amountOutMinimum;
}
```

As a result, any call to `Bera_Module::bera_kodiakv3_swap` will always revert due to the mismatched function parameters.

Impact: Swaps executed through Kodiak V3 will fail entirely, as every transaction using `bera_kodiakv3_swap` will revert.

Proof of Concept: The following test will fail and highlights the issue:

```
function test_bera_kodiakv3_swap() public {
    deal(WBERA_ADDRESS, address(strategy), 1e18);

    bytes memory path = abi.encodePacked(WBERA_ADDRESS, uint24(3000), HONEY_ADDRESS);

    vm.prank(EXECUTOR);
    bera.bera_kodiakv3_swap(
        WBERA_ADDRESS,
        1e18,
        0,
        path
    );

    assertGt(HONEY.balanceOf(address(strategy)),0);
}
```

Recommended Mitigation: Consider removing the `deadline` parameter from `IKodiakV3.ExactInputParams`. Also, consider adding a separate `deadline` that can be passed to the call to ensure proper deadline handling, as discussed in issue *7.4.3 Improper deadline handling*.

D2: Fixed in 84dbcf9

Cyfrin: Verified.

7.3.7 Kodiak swap functions do not check if output token is approved, risking stuck assets

Description: In the swap implementations within `Inch_Module`, both the input token and output token are validated to ensure they are approved before executing a swap. This validation is present in inch_swap, inch_uniswapV3Swap, and inch_clipperSwap.

However, this validation is missing in both `Bera_Module::bera_kodiakv2_swap` and `Bera_Module::bera_kodiakv3_swap`.

As a result, the trader (`EXECUTOR_ROLE`) could accidentally or intentionally swap to a non-approved token. Since this token **cannot be swapped back, it would remain stuck in the Strategy contract, making it inaccessible.

Impact: If a trader swaps into a non-approved token, it would become irreversibly stuck in the contract, as it cannot be swapped back.

Proof of Concept: The following test highlights the issue:

```
function test_bera_kodiakv2_swap_to_non_allowed_token() public {
    deal(WBERA_ADDRESS, address(strategy), 1e18);

    address[] memory path = new address[](2);
    path[0] = WBERA_ADDRESS;
    path[1] = RAMEN_TOKEN_ADDRESS; // not allowed

    vm.startPrank(EXECUTOR);
    bera.bera_kodiakv2_swap(
        WBERA_ADDRESS,
        1e18,
        0,
        path
    );

    uint256 ramenBalance = ERC20(RAMEN_TOKEN_ADDRESS).balanceOf(address(strategy));
    assertGt(ramenBalance,0);

    path[0] = RAMEN_TOKEN_ADDRESS;
    path[1] = WBERA_ADDRESS;

    // token cannot be swapped back
    vm.expectRevert("Invalid token");
    bera.bera_kodiakv2_swap(
        RAMEN_TOKEN_ADDRESS,
        ramenBalance,
        0,
        path
    );
    vm.stopPrank();

    // balance still left in contract
    assertEq(ERC20(RAMEN_TOKEN_ADDRESS).balanceOf(address(strategy)), ramenBalance);
}
```

Recommended Mitigation: Consider validating both input token *and* output token. As is done in the `Inch_Module` swaps.

D2: Fixed in `c70268c`

Cyfrin: Verified.

7.4 Low Risk

7.4.1 Aave's module lacks reward-claiming functionality

Description: The D2 protocol's Aave integration lacks reward claiming functionality in the Aave_Module.sol contract. While the protocol successfully interacts with Aave's lending and borrowing features, it does not implement any mechanism to claim or distribute rewards earned from these activities.

The contract lacks both the interface definition and implementation for interacting with Aave's IncentivesController, which is necessary for claiming rewards. As a result, any rewards accrued from lending and borrowing activities remain locked in the Aave protocol.

Impact: The absence of reward claiming functionality leads to several consequences:

1. **Permanent Loss of Value**
   - All rewards earned from lending and borrowing activities are effectively locked in the Aave protocol
   - These rewards accumulate but remain inaccessible to both users and the protocol
   - The value is permanently stranded unless reward claiming functionality is added
2. **Reduced Protocol Revenue**
   - The protocol misses out on potential revenue streams from reward tokens
   - Lost opportunity for treasury diversification through reward tokens

Recommended Mitigation: To address this, consider implementing:

1. Integration with Aave's IncentivesController to enable reward claiming
2. A reward distribution system that fairly allocates claimed rewards between users and protocol
3. Regular automated claiming mechanisms to optimize reward collection

D2: Fixed in `d3f76bb`

Cyfrin: Verified.

7.4.2 Compromised trader account can block admin from revoking access

Description: The trading strategy contract for D2 defines two roles: `ADMIN_ROLE` and `EXECUTOR_ROLE`. The `EXECUTOR_ROLE` is intended for the active party responsible for executing trades on behalf of stakers, while the `ADMIN_ROLE` is a more passive role, primarily reserved for emergency interventions.

The issue arises in `Vault::initialize`, where the `args.trader` is assigned both `ADMIN_ROLE` and `EXECUTOR_ROLE`:

```
s.grantRole(ADMIN_ROLE, args._owner);
s.grantRole(EXECUTOR_ROLE, args._owner);
s.grantRole(ADMIN_ROLE, args._trader); // @audit admin given to trader as well
s.grantRole(EXECUTOR_ROLE, args._trader);
```

This setup introduces a security risk: if the trader's account is compromised, the attacker could revoke the owner's admin privileges, preventing them from removing the compromised trader from the `EXECUTOR_ROLE`.

Impact: If the trader account is compromised, the protocol admin may be unable to revoke the `EXECUTOR_ROLE` from the attacker before they can exploit their access. This could result in unauthorized or malicious trades, potentially harming the protocol and its stakeholders.

Recommended Mitigation: Consider not assigning `ADMIN_ROLE` to `args.trader`:

```
  s.grantRole(ADMIN_ROLE, args._owner);
  s.grantRole(EXECUTOR_ROLE, args._owner);
- s.grantRole(ADMIN_ROLE, args._trader);
  s.grantRole(EXECUTOR_ROLE, args._trader);
```

D2: Fixed in 614daaa

Cyfrin: Verified.

7.4.3 Improper deadline handling

Description: The Bera module shows improper handling of transaction deadlines in DEX operations:

- KodiakV2/V3 operations use `type(uint256).max` as deadline:

```
// Bera.sol
function bera_kodiakv2_add(...) {
    kodiakv2.addLiquidity(..., type(uint256).max);
}

function bera_kodiakv2_swap(address token, uint amount, uint amountMin, address[] calldata path)
↪   external onlyRole(EXECUTOR_ROLE) nonReentrant {
    ...
    kodiakv2.swapExactTokensForTokensSupportingFeeOnTransferTokens(
        ...
        type(uint256).max
    );
}

function bera_kodiakv3_increase(...) {
    kodiakv3.increaseLiquidity(IKodiakV3.IncreaseLiquidityParams({
        ...
        deadline: type(uint256).max
    }));
}
```

- OogaBooga and KodiakV3 (see issue *7.3.6 `Bera_Module::bera_kodiakv3_swap` broken due to `deadline` parameter*) swaps lack deadline protection entirely:

```
function bera_oogabooga_swap(...) {
    oogaBooga.swap(tokenInfo, pathDefinition, executor, referralCode);
}
```

Using infinite deadlines or no deadlines at all is a risky practice in DeFi. Transaction deadlines are a crucial protection mechanism that prevents trades from executing under unexpected market conditions, especially during network congestion or high volatility periods.

Impact: While access is restricted to EXECUTOR_ROLE, improper deadline handling could lead to:

- Trades executing at unexpected times during network congestion
- No protection against stale transactions in volatile market conditions
- MEV bots could hold transactions and execute them at disadvantageous moments
- No mechanism to invalidate outdated trades

For OogaBooga specifically, the lack of deadline parameter in their protocol design leaves users exposed to these risks without any way to protect themselves.

Recommended Mitigation:

- For Kodiak operations, implement reasonable deadlines:

```
uint256 private constant MAX_DEADLINE_WINDOW = 30 minutes;

function bera_kodiakv2_add(...) {
    kodiakv2.addLiquidity(..., block.timestamp + MAX_DEADLINE_WINDOW);
}
```

- For OogaBooga and Kodiak V3, consider implementing a wrapper that adds deadline protection:

```
function bera_oogabooga_swap(..., uint256 deadline) {
   require(block.timestamp <= deadline, "Expired");
    oogaBooga.swap(...);
}

function bera_kodiakv3_swap(..., uint256 deadline) external onlyRole(EXECUTOR_ROLE) nonReentrant {
    require(block.timestamp <= deadline, "Expired");
    ...
    kodiakv3swap.exactInput(IKodiakV3.ExactInputParams({
        ...
    }));
}
```

D2: Ignored "Improper deadline handling" because we mostly don't care about it on our main chain Arbitrum, although we understand how it's riskier for on Mainnet

Cyfrin: Acknowledged.

7.4.4 Missing events for important state changes

Description: The following calls lack events emitted even though they change important states inside the contracts:

- `Strategy::claim`
- `Strategy::setFrozen`
- `Strategy::setSelector`
- `VaultV3::emergencyFreeze`

Recommended Mitigation: Consider adding events to be emitted in the above calls. This provides a clear on-chain record of when and by whom the strategy was claimed, improving transparency and auditability.

D2: Ignored "Missing events for important state changes" as we will remove those calls shortly

Cyfrin: Acknowledged.

7.4.5 `Silo_Module::silo_execute` will revert as approval is to the wrong contract

Description: D2 integrates with the lending protocol Silo via the `Silo_Module`, which facilitates communication with the Silo contract.

One of the functions in this module is `Silo_Module::silo_execute`, which allows bundling multiple calls and sending them to the Silo Router to optimize gas usage:

```
function silo_execute(ISiloRouter.Action[] calldata actions) external onlyRole(EXECUTOR_ROLE)
↪   nonReentrant {
    // @audit approval should be to `address(router)` as it is the one facilitating the actions
    IERC20(actions[0].asset).approve(actions[0].silo, actions[0].amount);
    router.execute(actions);
}
```

The issue here is that the approval is incorrectly granted to `actions[0].silo`, whereas it should be granted to `router`, as this is the contract responsible for handling all token transfers.

Impact: The `silo_execute` function will fail due to the incorrect approval. While this does not affect the overall functionality of `Silo_Module`—since all necessary actions can still be executed separately—the gas savings intended by bundling transactions will be lost, leading to higher transaction costs.

Proof of Concept: The following test highlights the issue:

```
function test_silo_execute() public {
    deal(WETH_ADDRESS,address(strategy),1e18);

    ISiloRouter.Action[] memory actions = new ISiloRouter.Action[](4);
    actions[0] = ISiloRouter.Action(ISiloRouter.ActionType.Deposit, GMX_SILO, WETH_ADDRESS, 1e18,
    ↪  false);
    actions[1] = ISiloRouter.Action(ISiloRouter.ActionType.Borrow, GMX_SILO, GMX_ADDRESS, 100e18,
    ↪  false);
    actions[2] = ISiloRouter.Action(ISiloRouter.ActionType.Repay, GMX_SILO, GMX_ADDRESS, 100e18, false);
    actions[3] = ISiloRouter.Action(ISiloRouter.ActionType.Withdraw, GMX_SILO, WETH_ADDRESS, 1e18-1,
    ↪  false);

    vm.prank(EXECUTOR);
    silo.silo_execute(actions);

    assertEq(gmxDebtToken.balanceOf(address(strategy)), 1);
    assertEq(gmxCollateralToken.balanceOf(address(strategy)), 0);
    assertEq(wethCollateralToken.balanceOf(address(strategy)), 0);
    assertEq(address(strategy).balance, 1e18-1);
}
```

Recommended Mitigation: Consider approving `router` instead:

```
- IERC20(actions[0].asset).approve(actions[0].silo, actions[0].amount);
+ IERC20(actions[0].asset).approve(address(router), actions[0].amount);
```

D2: Fixed in ea03f4d

Cyfrin: Verified.

7.4.6 Silo and Dolomite lack LTV limit increasing liquidation risk

Description: When borrowing from Aave, there is a check in `Aave_Module::aave_borrow` to ensure that the loan-to-value (LTV) ratio remains below `MAX_LTV_FACTOR` (80%):

```
pool.borrow(asset, amount, interestRateMode, referralCode, onBehalfOf);
(uint256 totalCollateralBase, uint256 totalDebtBase, , , uint256 ltv, ) =
↪  pool.getUserAccountData(onBehalfOf);
uint256 maxDebtBase = (totalCollateralBase * ltv * MAX_LTV_FACTOR) / (BASIS_FACTOR * MANTISSA_FACTOR);
require(totalDebtBase <= maxDebtBase, "borrow amount exceeds max LTV");
```

However, this LTV check is missing from `Silo_Module::silo_borrow` and `Dolomite_Module::dolomite_openBorrowPosition`/`dolomite_transferBetweenAccounts`, which are also lending protocols. As a result, positions with a higher LTV than 80% can be opened in these platforms.

Impact: Since Silo and Dolomite do not enforce the same 80% LTV limit higher risk positions can be entered. This increases the likelihood of unexpected liquidations, exposing users to unnecessary financial risk.

Recommended Mitigation: Consider implementing the same LTV checks for Silo and Dolomite.

D2: Ignored "Silo and Dolomite lack LTV limit increasing liquidation risk", will assume the trader is not reckless

Cyfrin: Acknowledged.

7.4.7 Borrowing non-approved tokens can bypass trading restrictions

Description: In the `Inch_Module` swap functions, the output token is verified to ensure it is approved by the protocol. This validation helps maintain trust that only pre-approved tokens will be traded.

However, the trader can still borrow non-approved tokens, bypassing this restriction. This undermines the assumption that only approved tokens can be used or held by the Strategy contract.

Impact: The expectation that only approved tokens will be used in trading can be violated if a trader borrows and holds non-approved tokens. This weakens the protocol's asset control and may introduce unforeseen risks.

Recommended Mitigation: Consider validating that the borrowed tokens in Aave, Silo and Dolomite are also approved tokens.

D2: Fixed for Aave and Silo in `1704b31`, Dolomite acknowledged.

Cyfrin: Verified.

7.5 Informational

7.5.1 `ETH` calls on `Dolomite_Module` have no corresponding calls on Dolomite

Description: The following calls have no corresponding call on the Dolomite deployed contract:

- `Dolomite_Module::dolomite_depositETH`
- `Dolomite_Module::dolomite_withdrawETH`
- `Dolomite_Module::dolomite_depositETHIntoDefaultAccount`
- `Dolomite_Module::dolomite_withdrawETHFromDefaultAccount`

Consider removing them.

D2: Removed in `dc68fe8`

Cyfrin: Verified.

7.5.2 Unnecessary validation in `Bera_Module::bera_infrared_stake` should follow standard pattern

Description: In `Bera_Module::bera_infrared_stake`, there is a call to `validateToken(vault)`:

```
function bera_infrared_stake(address vault, address token, uint256 amount) external
↪  onlyRole(EXECUTOR_ROLE) nonReentrant {
    validateToken(vault); // @audit unnecessary
    IERC20(token).approve(vault, amount);
    IInfraredVault(vault).stake(amount);
}
```

This is unusual because the validation is applied to `vault` rather than `token`. Additionally, the check is unnecessary since the `token` must already be present in the contract, and for that to happen, it must be included in `allowedTokens`.

However, the `validateToken(vault)` function does serve a purpose—it prevents the trader from using a malicious `vault` contract. Therefore, the target should still be validated, but using a more appropriate approach. This pattern is already established in other parts of the codebase, where the validation is performed inside the `TraderV0::approve` function. To maintain consistency, we suggest following the same approach here.

Consider removing both the `validateToken(vault);` call and the approval altogether:

```
function bera_infrared_stake(address vault, address token, uint256 amount) external
↪  onlyRole(EXECUTOR_ROLE) nonReentrant {
-   validateToken(vault);
-   IERC20(token).approve(vault, amount);
    IInfraredVault(vault).stake(amount);
}
```

D2: Fixed in `b0c4c4b`

Cyfrin: Verified.

7.5.3 Unused imports

Description: There are a couple of unused imports in the contracts:

```
import "@solidstate/contracts/introspection/ERC165/base/ERC165Base.sol";
import "@solidstate/contracts/proxy/diamond/writable/DiamondWritableInternal.sol";
```

These are imported in:

- `Dolomite.sol#L6-L7`

- Dolomite.sol#L6-L7, Here they're used by GMXV2_Cutter but since this abstract contract is also unused it can also be removed.
- Pendle.sol#L6-L7
- Silo.sol#L6-L7
- WETH.sol#L6-L7

Also, the Ownable import in D2OFT.sol#L4 is not used:

```
import "@openzeppelin/contracts/access/Ownable.sol";
```

Consider removing these.

D2: Removed in 8880a91

Cyfrin: Verified.

7.5.4 Aave swapBorrowRateMode is deprecated

Description: In the Aave_Module there is a call aave_swapBorrowRateMode:

```
function aave_swapBorrowRateMode(address asset, uint256 interestRateMode) external
↪  onlyRole(EXECUTOR_ROLE) nonReentrant {
    pool.swapBorrowRateMode(asset, interestRateMode);
}
```

The call swapBorrowRateMode is however deprecated, it only exists as a call on the v2 pool.

Consider removing this call.

D2: Removed in cbcddab

Cyfrin: Verified.

7.5.5 Selectors for Bera_Module::bera_kodiakv2_swap and bera_kodiakv3_swap needs to be separately added

Description: In Strategy::constructor almost all existing facet selectors are added to the s.selectors mapping, which is then used by the fallback function to direct the call to the correct facet.

Almost all, the Bera_Module::bera_kodiakv2_swap and bera_kodiakv3_swap are not present here and need to be added separately using the Strategy::setSelector call limited to role 0x00 (DEFAULT_ADMIN_ROLE).

Consider including these in the general selector assignment in Strategy::constructor instead. Thus saving two calls.

D2: Added in 5e22afc

Cyfrin: Verified.

7.5.6 Consider using Ownable2Step instead of Ownable

Description: The VaultV3 and VaultV0 contracts use OpenZeppelin's Ownable for access control. While secure, this single-step ownership transfer pattern risks permanently locking the contracts if ownership is accidentally transferred to an invalid or inaccessible address.

Consider using Ownable2Step instead, which requires the new owner to accept ownership in a separate transaction. This two-step pattern is considered best practice as it prevents accidental transfers and ensures the new owner can actually interact with the contract.

D2: Ignored "Consider using Ownable2Step instead of Ownable" we'll consider it for the future

Cyfrin: Acknowledged.