

# Dezy Contracts V2















## Security Testing and Assessment

August 24th, 2023

*Prepared for Dezy Finance*

# Table of Contents

<b>Table of Contents</b>	<b>2</b>
<b>Summary</b>	<b>8</b>
Overview	8
Project Scope	8
Summary of Findings	8
<b>Disclaimer</b>	<b>9</b>
<b>Key Findings and Recommendations</b>	<b>10</b>
1. No slippage protection in UniswapV2	10
Description	10
Impact	11
Recommendation	11
2. The swap deadline is not correctly set	12
Description	12
Impact	12
Recommendation	13
Remediation	13
3. The AaveV3 module is vulnerable to potential sandwich attack	14
Description	14
Impact	14
Recommendation	15
Remediation	15
4. Missing LibuniswapV3Mapping update when calling decreaseLiquidity()	16
Description	16
Impact	16
Proof of Concept	17
Recommendation	17
Remediation	17
5. Incorrect index update in LibSmartWalletAuth	18
Description	18
Impact	19
Proof of Concept	19
Recommendation	19
Remediation	19
6. Potential reentrancy in SmartWalletAA	20
Description	20
Impact	20

Recommendation	20
Remediation	20
7. Incorrect funcSig value in SmartWalletMulticall	21
Description	21
Impact	21
Recommendation	22
Remediation	22
8. Unmatched docstring in LibReentrancyGuard	23
Description	23
Recommendation	23
Remediation	23
9. Zero should not be used as the magic value to represent the entire balance	24
Description	24
Impact	24
Recommendation	24
Remediation	25
<b>Testing Specifications and Results</b>	<b>26</b>
Dezy V2	26
Property Tests	26
Contract: SmartWalletDiamondFactory	26
testCreateWallet 	26
Contract: SmartWalletAA	26
testFailExecOpFor 	26
testExecOpFor 	26
testExecOpForBlacklist 	27
Contract: SmartWalletMulticall	27
testExecuteFlow 	27
testExecuteFlowWithReferral 	27
testExecuteFlow2 	27
Contract: ProtocolFee	28
testProtocolFee 	28
testCalcProtocolFee 	28
testClaim 	28
Contract: Verifier	29
testVerify 	29
Contract: DCAStrategy	29
testDepositFillWithdraw 	29
Contract: DezyDca	29
testDezyDeposit 	29
testDezyWithdraw 	30
Contract: AaveV3	30

testAaveDeposit	✓	30
testAaveWithdraw	✓	30
testAaveBorrow	✓	31
Contract: Cega		31
testCegaDeposit	✓	31
testCegaDepositLeverage	✓	31
testCegaWithdrawal	✓	32
Contract: GearboxEarn		32
testGearboxEarnDeposit	✓	32
testGearboxEarnWithdraw	✓	32
Contract: InstaDapp		33
testInstaDappDeposit	✓	33
testInstaDappWithdraw	✓	33
Contract: Lido		33
testLidoDeposit	✓	33
testLidoWithdraw	✓	34
testLidoClaim	✓	34
testLidoWrap	✓	35
Contract: RocketPool		35
testRocketPoolDeposit	✓	35
testRocketPoolWithdraw	✓	35
Contract: Unagi		36
testUnagiDeposit	✓	36
testUnagiWithdraw	✓	36
Contract: Vesper		36
testVesperDeposit	✓	36
testVesperWithdraw	✓	37
testVesperClaim	✓	37
Contract: VesperGovernance		38
testVesperGovernanceDeposit	✓	38
testVesperGovernanceWithdraw	✓	38
Contract: UniswapV2		38
testUniswapV2Deposit	✓	38
testUniswapV2Withdraw	✓	39
Contract: UniswapV3		39
testUniswapV3Deposit	✓	39
testUniswapV3Withdraw	✓	39
testUniswapV3Claim	✓	40
testUniswapV3WithdrawNFT	✓	40
testUniswapV3UpdatePosition	✓	40
Contract: NotionalLP		41

testNotionalLPDeposit	✓	41
testNotionalLPWithdraw	✓	41
testNotionalLPClaim	✓	41
Contract: NotionalFD		42
testNotionalFDDeposit	✓	42
testNotionalFDWithdraw	✓	42
Invariant Tests		43
Contract: Diamond		43
invariantDiamondOwner	✓	43
Contract: Admin		43
invariantDiamondAdmin	✓	43
Contract: ProtocolFee		43
invariantFeeConfig	✓	43
Contract: FlashLoanHandler		44
invariantFlashloanHandlerCallerAndInput	✓	44
Contract: Nft		44
invariantNFTHandler	✓	44
Contract: SmartWalletAuth		44
invariantWhitelistFn	✓	44
invariantBlacklistFn	✓	45
Contract: FeeCollector		45
invariantOwner	✓	45
invariantOwnerClaim	✓	45
invariantUserClaim	✓	46
Contract: AaveV3		46
invariantDepositWithdraw	✓	46
invariantUserProfitRangeBroken	✓	46
Contract: Cega		46
invariantDepositWithdraw	✓	46
invariantUserProfitRangeBroken	✓	47
Contract: GearboxEarn		47
invariantDepositWithdraw	✓	47
invariantUserProfitRangeBroken	✓	47
Contract: InstaDapp		48
invariantDepositWithdraw	✓	48
invariantUserProfitRangeBroken	✓	48
Contract: Lido		48
invariantDepositWithdraw	✓	48
invariantUserProfitRangeBroken	✓	48
Contract: NotionalFD		49
invariantDepositWithdraw	✓	49

invariantUserProfitRangeBroken ✓	49
Contract: NotionalLP	49
invariantDepositWithdraw ✓	49
invariantUserProfitRangeBroken ✓	50
Contract: RocketPool	50
invariantDepositWithdraw ✓	50
invariantUserProfitRangeBroken ✓	50
Contract: Unagii	50
invariantDepositWithdraw ✓	50
invariantUserProfitRangeBroken ✓	51
Contract: UniswapV2	51
invariantDepositWithdraw ✓	51
invariantUserProfitRangeBroken ✓	51
Contract: UniswapV3	52
invariantDepositWithdraw ✓	52
invariantUserProfitRangeBroken ✓	52
Contract: Vesper	52
invariantDepositWithdraw ✓	52
invariantUserProfitRangeBroken ✓	52
Contract: VesperGovernance	53
invariantDepositWithdraw ✓	53
invariantUserProfitRangeBroken ✓	53
Contract: DezyDCA	53
invariantDepositWithdraw ✓	53
invariantUserProfitRangeBroken ✓	54
invariantFees ✓	54
invariantPause ✓	54
Contract: Fees	54
invariantFees ✓	54
invariantDepositStatus ✓	55
invariantFeeCollector ✓	55
invariantDefaultFee ✓	55
invariantWhitelistTokens ✓	55
Contract: OracleAggregator	56
invariantChainlinkTokens ✓	56
invariantUniswapPools ✓	56
<b>Relevant Past Hacks</b>	<b>57</b>
Address misconfiguration	57
Yearn Finance	57
Arbitrary unstake	57
Thena	57

Wrong accounting	57
Spherax USDs	57
Custom ERC20 implementation	58
Thoreum Finance	58
The calculation of rewards depended on the caller's balance	58
NewFreeDAO	58
Reentrancy	58
Paraluni	58
Price Oracle manipulation	59
RodeoFinance	59
Ovix	59
BonqDAO	59
Inverse Finance	59
Public price setting	59
Fortress Loans	59
Public set price oracle	60
Rikkei Finance	60
<b>Fix Log</b>	<b>60</b>
<b>Appendix</b>	<b>61</b>
Severity Categories	62

# Summary

## Overview

From June 21, 2023, to August 4, 2023, Dezy engaged Narya.ai to evaluate the security of its DezyV2 contracts in the GitHub repository: <https://github.com/DezyDefi/contracts-v2-audit> (commit [d572f5b8786aed10a9f97b683f17720b435f41c1](https://github.com/DezyDefi/contracts-v2-audit/commit/d572f5b8786aed10a9f97b683f17720b435f41c1)).

The main changes since the V1 include the following:

- Refactoring into the diamond pattern.
- Strategies are facets of the main diamond.
- User wallets refactoring into diamonds instead of proxies.
- Signature verification for users acting on behalf of the wallet owner.

## Project Scope

We reviewed and tested the DezyV2 contracts. Other security-critical components of DezyV2, such as off-chain services and the web front end, are not included in the scope of this security assessment. We recommend a further review of those components.

## Summary of Findings

Severity	# of Findings
High	0
Medium	5
Low	2
Informational	2
Gas	0
Total	9

We have implemented 49 property tests and 47 invariants to test the following key logic for the DezyV2 contracts.

**We have found 9 issues throughout the testing. All the tests have been using the latest commit [d572f5b8786aed10a9f97b683f17720b435f41c1259a228321250ff8f](https://github.com/DezyDefi/contracts-v2-audit/commit/d572f5b8786aed10a9f97b683f17720b435f41c1259a228321250ff8f).**



# Disclaimer

This report should not be used as investment advice.

Narya.ai uses an automatic testing technique to test smart contracts' security properties and business logic rapidly and continuously. However, we do not provide any guarantees on eliminating all possible security issues. The technique has its limitations: for example, it may not generate a random edge case that violates an invariant during the allotted time. Its use is also limited by time and resource constraints.

Unlike time-boxed security assessment, Narya.ai advises continuing to create and update security tests throughout the project's lifetime. In addition, Narya.ai recommends proceeding with several other independent audits and a public bug bounty program to ensure smart contract security.

# Key Findings and Recommendations

## 1. No slippage protection in UniswapV2

Severity: **Medium**

### Description

Front-running is possible in Uniswap V2 and Uniswap V3 strategies, in the withdrawal logic. The minimum amounts should not be set to 0.

**Code 1.1** [implementation/Ethereum/contracts/DezyModules/Uniswap/UniswapV2.sol#L119](#)

```
function uniswapV2Withdraw(
    address poolAddress,
    uint256 amount
) public payable returns (TokenAmt[] memory) {
    IUniswapV2Pair pool = IUniswapV2Pair(poolAddress);
    address token0 = pool.token0();
    address token1 = pool.token1();

    uint amountA;
    uint amountB;

    LibSafeApprove.safeApprove(poolAddress, uniV2Router, amount);
    (amountA, amountB) =
    IUniswapV2Router(uniV2Router).removeLiquidity(token0, token1, amount, 0, 0,
    address(this), block.timestamp);
    ...
}
```

**Code 1.2** [implementation/Ethereum/contracts/DezyModules/Uniswap/UniswapV3.sol#L384](#)

```
function _withdraw(
    address poolAddress,
    uint128 amount
) internal returns(uint, uint, address, address) {
    IUniswapV2Pair pool = IUniswapV2Pair(poolAddress);
    address token0 = pool.token0();
    address token1 = pool.token1();

    uint balanceA = IERC20(token0).balanceOf(address(this));
    uint balanceB = IERC20(token1).balanceOf(address(this));

    uint24 poolFee = IUniswapV3Pool(poolAddress).fee();
}
```

```
uint256 nftId = uniswapV3GetNftId(
    token0,
    token1,
    poolFee
);

INonfungiblePositionManagerProxy.DecreaseLiquidityParams memory
params = INonfungiblePositionManagerProxy.DecreaseLiquidityParams({
    tokenId: nftId,
    liquidity: amount,
    amount0Min: 0,
    amount1Min: 0,
    deadline: block.timestamp
});
...

```

## Impact

The user may suffer from high slippage due to a sandwich attack.

## Recommendation

It is recommended to have the minimum amounts passed from the function arguments to check against slippage.

## Remediation

This was fixed at commit [7c8634b8fca1288ba1c94444cf513c2bed6aeba0](#).

## 2. The swap deadline is not correctly set

Severity: **Medium**

### Description

Hard-coding the deadline to `block.timestamp` makes the transaction never out-of-date.

The following lines directly call `approve()` with an arbitrary ERC20 token:

- [implementation/Ethereum/contracts/DezyModules/Uniswap/UniswapSwap.sol#L59](#)
- [implementation/Ethereum/contracts/DezyModules/Uniswap/UniswapSwap.sol#L108](#)
- [implementation/Ethereum/contracts/DezyModules/Uniswap/UniswapV2.sol#L81](#)
- [implementation/Ethereum/contracts/DezyModules/Uniswap/UniswapV2.sol#L119](#)
- [implementation/Ethereum/contracts/DezyModules/Uniswap/UniswapV3.sol#L250](#)
- [implementation/Ethereum/contracts/DezyModules/Uniswap/UniswapV3.sol#L263](#)
- [implementation/Ethereum/contracts/DezyModules/Uniswap/UniswapV3.sol#L384](#)

For example,

**Code 2** [implementation/Ethereum/contracts/DezyModules/Uniswap/UniswapSwap.sol#L108](#)

```
function uniswapSwapV2ExactOutput(
    address[] memory path,
    uint256 amountOut,
    uint rate,
    uint slippage
) public payable returns (TokenAmt[] memory) {
    uint amountInMax = (((amountOut / rate) * (DIVISOR + slippage)) /
DIVISOR) * 10 ** IERC20(path[0]).decimals();
    LibSafeApprove.safeApprove(path[0], uniswapV2Router, amountInMax);
    uint[] memory amountsIn =
IUniswapV2Router(uniswapV2Router).swapTokensForExactTokens(
        amountOut,
        amountInMax,
        path,
        address(this),
        block.timestamp
    );
    ...
}
```

### Impact

The user may have to spend up to the maximum input amount due to a sandwich attack.

## Recommendation

It's recommended to add an additional deadline argument.

## Remediation

This was fixed at commit [7c8634b8fca1288ba1c94444cf513c2bed6aeba0](#).

### 3. The AaveV3 module is vulnerable to potential sandwich attack

Severity: **Medium**

#### Description

In `aaveV3RepayWithFlashLoanExecuteOperation`, the `maxCollateralToWithdraw` collateral tokens can be used to repay the flash loan. Consider the case when the user deposits 100 ETH and borrows 1 USD, then when the user is repaying the 1 USD a swap is performed and subject to a sandwich attack.

[Code 3 implementation/Ethereum/contracts/DezyModules/Aave/AaveV3.sol#L412](#)

```
function aaveV3RepayWithFlashLoanExecuteOperation(
    address asset,
    uint256 amount,
    uint256 premium,
    bytes calldata params
) public payable returns (bool) {
    (
        /* bytes4 selector */,
        address collateral,
        uint debtToRepay,
        uint interestRateMode,
        address onBehalfOf
    ) = abi.decode(params, (bytes4, address, uint, uint, address));
    // Aave loan repay
    _aaveV3Repay(asset, debtToRepay, interestRateMode, onBehalfOf);
    // Aave collateral withdraw
    uint maxCollateralToWithdraw =
        calculateMaxCollateralToWithdraw(collateral, onBehalfOf);
    IAavePool.ReserveData memory collateralReserveData =
        IAavePool(aaveV3).getReserveData(collateral);

    IERC20(collateralReserveData.aTokenAddress).transferFrom(onBehalfOf,
        address(this), maxCollateralToWithdraw);
    _aaveV3Withdraw(collateral, maxCollateralToWithdraw);

    // Swapping part of the collateral to repay the flash loan fee
    IERC20(collateral).approve(swapRouter, maxCollateralToWithdraw);
    uint flashLoanAmount = amount + premium;
    ISwapRouter(swapRouter).swapTokenForTokenV3ExactOutput(
        collateral,
```

```
        asset,  
        flashLoanAmount,  
        maxCollateralToWithdraw,  
        address(this)  
    );  
  
    // flash loan repay approval handled by flashloan handler  
    return true;  
}
```

## Impact

The user may have to spend all the collateral due to a sandwich attack.

## Recommendation

It's recommended to involve a price oracle or add an additional off-chain provided argument to set a proper value for the `_amountInMaximum` argument of the swap.

## Remediation

This was fixed at commit [3e9b4937af925d4b5687d2727562a2a1b8555da9](#).

## 4. Missing LibuniswapV3Mapping update when calling decreaseLiquidity()

Severity: **Medium**

### Description

In the `uniswapV3Deposit` function, `LibuniswapV3Mapping.setPositionLiquidity` is called when calling `mint()` or `increaseLiquidity()`.

However, in the `uniswapV3Withdraw` function, the `setPositionLiquidity` function is not called correspondingly.

[Code 4 implementation/Ethereum/contracts/DezyModules/Uniswap/UniswapV3.sol#L361](#)

```
function _withdraw(
    address poolAddress,
    uint128 amount
) internal returns(uint, uint, address, address) {
    IUniswapV2Pair pool = IUniswapV2Pair(poolAddress);
    address token0 = pool.token0();
    address token1 = pool.token1();

    uint balanceA = IERC20(token0).balanceOf(address(this));
    uint balanceB = IERC20(token1).balanceOf(address(this));

    uint24 poolFee = IUniswapV3Pool(poolAddress).fee();
    uint256 nftId = uniswapV3GetNftId(
        token0,
        token1,
        poolFee
    );

    INonfungiblePositionManagerProxy.DecreaseLiquidityParams memory
    params = INonfungiblePositionManagerProxy.DecreaseLiquidityParams({
        tokenId: nftId,
        liquidity: amount,
        amount0Min: 0,
        amount1Min: 0,
        deadline: block.timestamp
    });
    INonfungiblePositionManagerProxy(nfpm).decreaseLiquidity(params);
    uniswapV3Claim(token0, token1, nftId);
}
```



```
uint amountA = IERC20(token0).balanceOf(address(this)) - balanceA;
uint amountB = IERC20(token1).balanceOf(address(this)) - balanceB;

emit Event(this.uniswapV3Withdraw.selector, 0,
abi.encode(poolAddress, amount, nftId, amountA, amountB));

return (amountA, amountB, token0, token1);
}
```

## Impact

This will block `uniswapV3UpdatePosition()` from withdrawing as it is using the wrong liquidity.

## Proof of Concept

<https://github.com/DezyDefi/contracts-v2-audit/blob/narya-tests/implementation/Ethereum/test/narya/poc/UniV3LiquidityTest.t.sol>

## Recommendation

`LibuniswapV3Mapping.setPositionLiquidity` should be called accordingly in `uniswapV3Withdraw`.

## Remediation

This was fixed at commit [d807323a33a92024db261b7d3528f9500448c5c0](#).

## 5. Incorrect index update in LibSmartWalletAuth

Severity: **Medium**

### Description

The whitelist mapping stores the fnSig's index in the array whitelistFns **plus 1**. However, +1 is missing in the deleting logic.

[Code 5 implementation/Ethereum/contracts/CoreModules/SmartWallet/libraries/LibSmartWalletAuth.sol#L31](#)

```
/// @dev sets a function signature to be whitelisted
function _setFnWhitelist(bytes4 fnSig, bool state) internal {
    SmartWalletAuthState storage smartWalletAuthState =
diamondStorage();
    if (smartWalletAuthState.whitelist[fnSig] > 0 && state == true) {
        return;
    }
    if (smartWalletAuthState.whitelist[fnSig] == 0 && state == false) {
        return;
    }
    if (state == true) {
        // add

        /// @dev push first so index 0 will always be blank
        smartWalletAuthState.whitelistFns.push(fnSig);
        smartWalletAuthState.whitelist[fnSig] = smartWalletAuthState
            .whitelistFns
            .length;
    } else {
        // delete
        uint delIndex = smartWalletAuthState.whitelist[fnSig] - 1;
        uint lastIndex = smartWalletAuthState.whitelistFns.length - 1;
        if (delIndex != lastIndex) {
            // overwrite with last element of array
            smartWalletAuthState.whitelistFns[
                delIndex
            ] = smartWalletAuthState.whitelistFns[lastIndex];
            // update mapping
            smartWalletAuthState.whitelist[
                smartWalletAuthState.whitelistFns[lastIndex]
            ] = delIndex;
        }
        delete smartWalletAuthState.whitelist[fnSig];
    }
}
```

```

        // shrink array
        smartWalletAuthState.whitelistFns.pop();
    }
}

```

Considering the following case where foo and bar are added:

```

whitelistFns[0] = foo
whitelist[foo] = 1
whitelistFns[1] = bar
whitelist[bar] = 2

```

After deleting foo, the expected status should be:

```

whitelistFns[0] = bar
whitelist[bar] = 1

```

However, `whitelist[bar]` becomes 0 now.

## Impact

This also affects the blacklist. A blacklisted function will become not blacklisted due to this issue. And for a whitelisted function, it will become not whitelisted.

## Proof of Concept

<https://github.com/DezyDefi/contracts-v2-audit/blob/narya-tests/implementation/Ethereum/test/narya/poc/SetFnWhitelist.t.sol>

## Recommendation

- Changing

```

smartWalletAuthState.whitelist[
    smartWalletAuthState.whitelistFns[lastIndex]
] = delIndex;

```

to

```

smartWalletAuthState.whitelist[
    smartWalletAuthState.whitelistFns[lastIndex]
] = delIndex + 1;

```

- Referring to the implementation of [EnumerableSet](#).

## Remediation

This was fixed at commit [a8406ab8b6b14081e9edf8c36349d22193c8a2fc](#)

## 6. Potential reentrancy in SmartWalletAA

Severity: **Low**

### Description

The `_executeOp` function in the `SmartWalletAA` contract does not follow the Checks-Effects-Interactions pattern, which could result in reentrancy.

[Code 9 implementation/Ethereum/contracts/CoreModules/SmartWallet/SmartWalletAA.sol#L163](#)

```
function _executeOp(bytes memory data) internal {
    address facet =
    IDezyDiamond(BaseModule.diamond).selectorToFacet(data);
    (bool success, ) = facet.delegatecall(data);
    if (!success) revert ExecOpFail();
    INonceManager(BaseModule.diamond).increaseNonce();
}
```

### Impact

If in the future a function without the `nonReentrant` modifier is introduced that transfers funds out and makes a callback without proper accounting, then the re-entrancy could be exploited to replay a signed message and drain funds.

### Recommendation

The nonce should be increased before the delegate-call.

### Remediation

This was fixed at commit [29000834856663b01750072d9a043a966a444295](#).

## 7. Incorrect funcSig value in SmartWalletMulticall

Severity: **Low**

### Description

In the `_executeFlow` function of the `SmartWalletMulticall` contract, `funcSig` is only updated when the condition `c.cubeConfig.length > 0` is true. Otherwise, the `funcSig` value is uninitialized which will be later used by the `_exec` function and the Action event.

**Code 10** [implementation/Ethereum/contracts/CoreModules/SmartWallet/SmartWalletMulticall.sol#L135](#)

```
function _executeFlow(
    bytes[] memory inx,
    FlowConfig[] memory config
) internal {
    address facet;
    bytes[] memory localStack = new bytes[](inx.length);
    FlowConfig memory c;
    CubeConfig[] memory cc;
    bytes memory data;
    bytes memory newData;
    bytes4 funcSig;
    bool success;
    TokenAmt[] memory tokenAmts;
    for (uint i; i < inx.length; ) {
        /// @dev checks config for override
        c = config[i];
        if (c.cubeConfig.length > 0) {
            cc = c.cubeConfig;
            // code to override parameters
            TokenAmt[] memory tokenData = new TokenAmt[](cc.length);
            /// @dev saves the lookback override data
            for (uint j; j < cc.length; ) {
                /// @dev j denotes the slot
                /// @dev lookBackIndex denotes the cube index to pull
                values from
                tokenAmts = abi.decode(
                    localStack[cc[j].cubeId],
                    (TokenAmt[])
                );
                /// @dev the override function will only override if
                amt > 0
                tokenData[j] = TokenAmt({
```

```

        token: tokenAmts[cc[j].index].token,
        amt: cc[j].cubeId < i ? tokenAmts[cc[j].index].amt
: 0

    });
    unchecked {
        ++j;
    }
}
/// @dev obtain overwritten instructions
(funcSig, data) = splitFunctionSelector(inx[i]);
(success, newData) = BaseModule.diamond.staticcall(
    abi.encodeWithSelector(c.oFuncSig, data, tokenData)
);
if (!success) revert ExecReadFail(success, newData);
(data) = abi.decode(newData, (bytes));
/// @dev we reattach the original funcsig & override the
instruction
    inx[i] = attachFunctionSelector(funcSig, data);
}
facet =
IDezyDiamond(BaseModule.diamond).selectorToFacet(inx[i]);
if (c.save) {
    /// @dev only store if save explicitly defined
    localStack[i] = _exec(facet, funcSig, i, inx[i]);
} else {
    _exec(facet, funcSig, i, inx[i]);
}
emit Action(funcSig, i, inx[i]);
unchecked {
    ++i;
}
}
}
}

```

## Impact

Code that relies on emitted events might be using the wrong function signature.

## Recommendation

Initialize funcSig with the function signature at the beginning of the function.

## Remediation

This was fixed at commit [200160870c2efb3951e2de98135398ca8b6b07d4](#).

## 8. Unmatched docstring in LibReentrancyGuard

Severity: **Informational**

### Description

The docstring of `LibReentrancyGuard.setOnlyOnceState()` does not match its implementation.

**Code 20** [implementation/Ethereum/contracts/CoreModules/ReentrancyGuard/libraries/LibReentrancyGuard.sol#L41](#)

```
/// @dev sets the protocol fee
function setOnlyOnceState(bytes4 fnSig, uint state) internal {
    ReentrancyGuardState storage reentrancyGuardState =
diamondStorage();
    reentrancyGuardState.onlyOnceState[fnSig][block.number] = state;
}
```

### Recommendation

Fix the documentation based on the actual code.

### Remediation

This was fixed at commit [af282e479ac534923f0077d97f32baa4dd093599](#).



## 9. Zero should not be used as the magic value to represent the entire balance

Severity: **Informational**

### Description

In the `transferOut` function of the `Transfer` contract, 0 is used as a magic value to indicate that the entire balance will be transferred out.

[Code 21 implementation/Ethereum/contracts/DezyModules/Common/Transfer.sol](#)

```
function transferOut(address token, uint amount) public payable {
    address owner = ISmartWalletDiamondFactory(BaseModule.diamond)
        .getUserFromWallet(address(this));
    if (owner == address(0)) revert Unauthorized();
    if (token == address(0)) {
        /// @dev use entire balance if amount is 0
        amount = amount == 0 ? address(this).balance : amount;
        (bool success, ) = owner.call{value: amount}("");
        if (!success) revert EthTransferFail();
    } else {
        /// @dev use entire balance if amount is 0
        amount = amount == 0
            ? IERC20(token).balanceOf(address(this))
            : amount;
        IERC20(token).transfer(owner, amount);
    }
}
```

This design choice introduces potential security risk: when this function is called with the `amount = 0` unintentionally (for example, the zero comes from the result of certain complicated calculations), transferring the entire balance is not expected and may lead to potential fund loss.

### Impact

Developers who are not familiar with the interface of the function may call it with zero amount unintentionally, which may result in fund loss at the end depending on the actual code logic.

### Recommendation

It is recommended to use a less common value like `type(uint256).max`.

## Remediation

This was fixed at commit [24c432e1a46767b318c567d187b3499b9e3e9733](#).

# Testing Specifications and Results

The test results are against the latest commit

[d572f5b8786aed10a9f97b683f17720b435f41c1259a228321250ff8f](https://github.com/DezyV2/SmartWalletDiamondFactory/commit/d572f5b8786aed10a9f97b683f17720b435f41c1259a228321250ff8f).

## Dezy V2

### Property Tests

Contract: SmartWalletDiamondFactory

testCreateWallet 

**Tested key logic:** Test that the wallet creation functionality is working no matter the number of users.

**Tested user journey:**

- Create wallets for multiple users.

**Test checks:**

- Check that the wallets can be created no matter the number of users.

Contract: SmartWalletAA

testFailExecOpFor 

**Tested key logic:** Test that only a correctly signed transaction can succeed.

**Tested user journey:**

- Sign a transaction with an arbitrary private key and deadline.
- User: execute the signed transaction.

**Test checks:**

- Check that the signed transaction is reverting.

testExecOpFor 

**Tested key logic:** Test that only a correctly signed transaction can succeed.

**Tested user journey:**

- WalletOwner1: sign a transaction for another user that targets a function that must succeed.
- User: execute the signed transaction.

**Test checks:**

- Check that the signed transaction is executed successfully.

testExecOpForBlacklist ❌

**Tested key logic:** Test that even with a valid signature a blacklisted call cannot be called.

**Tested user journey:**

- Blacklist the targeted function.
- WalletOwner1: sign a transaction for another user that targets the blacklisted function.
- User: execute the signed transaction.

**Test checks:**

- Check that the signed transaction reverts.

Contract: SmartWalletMulticall

testExecuteFlow ✅

**Tested key logic:** Test that multi-call works as intended and pay fees.

**Tested user journey:**

- WalletOwner1: sign a transaction for another user that does a multi-call and has a target function that should succeed.
- User: execute the signed transaction.

**Test checks:**

- Check that the signed transaction is executed successfully and the fees have been paid.

testExecuteFlowWithReferral ✅

**Tested key logic:** Test that multi-call works as intended and pay referral fees.

**Tested user journey:**

- WalletOwner1: sign a transaction for another user that does a referrer multi-call and has a target function that should succeed.
- User: execute the signed transaction.

**Test checks:**

- Check that the signed transaction is executed successfully and the referrer fees have been paid.

testExecuteFlow2 ✅

**Tested key logic:** Test a multi-user scenario where each user tries to deposit, withdraw and transfer funds out.

**Tested user journey:**

- WalletOwner1: deposit into a Uniswap V2 pool using a signed multi-call.

- walletOwner2: deposit into the same Uniswap V2 pool using a signed multi-call.
- WalletOwner1: withdraw from the Uniswap V2 pool using a signed multi-call.
- walletOwner2: withdraw from the Uniswap V2 pool using a signed multi-call.
- WalletOwner1: transfer tokens out using a signed multi-call.

**Test checks:**

- Check that the flow is successful and funds are withdrawn.

Contract: ProtocolFee

testProtocolFee 

**Tested key logic:** Test that the fee can be calculated correctly for any valid multiplier.

**Tested user journey:**

- diamondOwner: set protocol fee multiplier.
- Attempt to calculate the protocol fee.

**Test checks:**

- Check that the protocol fee calculation is successful.

testCalcProtocolFee 

**Tested key logic:** Test that the protocol fee and the referrer fee are successful for different amounts of gas spent.

**Tested user journey:**

- Attempt to calculate the protocol fee.
- Attempt to calculate the protocol fee with the referral.

**Test checks:**

- Check that the calculations are successful.

testClaim 

**Tested key logic:** Test that the claim function works for both the owner and the user.

**Tested user journey:**

- Attempt to claim the collected fees for the owner.
- Attempt to claim the collected fees for the referrer.

**Test checks:**

- Check that the claims are successful and correct.

Contract: Verifier

testVerify 

**Tested key logic:** Test that a signed message passes with all `verify` functions. As well as a random signed message will revert with any of the verify functions.

**Tested user journey:**

- WalletOwner1: sign a transaction for an arbitrary user.
- User: check that all verify functions in the verifier validate the signed transaction.
- User: check that a wrongly signed transaction cannot pass the verifier.

**Test checks:**

- Check that all verify functions work on a correctly signed transaction.
- Check that an invalid signed transaction cannot pass the verification check.

Contract: DCAStrategy

testDepositFillWithdraw 

**Tested key logic:** Test that the withdrawal returns the correct filled amounts.

**Tested user journey:**

- WalletOwner1: deposit WETH into the DCA strategy.
- WalletOwner2: deposit WETH into the DCA strategy.
- Fill the existing orders after an amount of time.
- WalletOwner1: withdraw USDC from the DCA strategy.

**Test checks:**

- Check that the withdrawal is successful.

Contract: DezyDca

testDezyDeposit 

**Tested key logic:** After 2 users do some operations (deposit/withdraw), the wallet owner should still be able to deposit.

**Tested user journey:**

- WalletOwner1: deposit some funds.
- WalletOwner2: withdraw some funds.
- WalletOwner2: deposit some funds.
- WalletOwner1: withdraw some funds.
- WalletOwner1: deposit some funds.

**Test checks:**

- Check that the deposit is successful.

testDezyWithdraw 

**Tested key logic:** After 2 users do some operations (deposit/withdraw), the wallet owner should still be able to withdraw if there are staked funds.

**Tested user journey:**

- WalletOwner1: deposit some funds.
- WalletOwner2: withdraw some funds.
- WalletOwner2: deposit some funds.
- WalletOwner1: withdraw some funds.
- WalletOwner1: deposit some funds.
- WalletOwner1: withdraw some funds.

**Test checks:**

- Check that the withdrawal is successful.

Contract: AaveV3

testAaveDeposit 

**Tested key logic:** After 2 users do some operations (deposit/withdraw), the wallet owner should still be able to deposit.

**Tested user journey:**

- WalletOwner1: deposit some funds.
- WalletOwner2: withdraw some funds.
- WalletOwner2: deposit some funds.
- WalletOwner1: withdraw some funds.
- WalletOwner1: deposit some funds.

**Test checks:**

- Check that the deposit is successful.

testAaveWithdraw 

**Tested key logic:** After 2 users do some operations (deposit/withdraw), the wallet owner should still be able to withdraw if there are staked funds.

**Tested user journey:**

- WalletOwner1: deposit some funds.
- WalletOwner2: withdraw some funds.
- WalletOwner2: deposit some funds.
- WalletOwner1: withdraw some funds.
- WalletOwner1: deposit some funds.
- WalletOwner1: withdraw some funds.

**Test checks:**

- Check that the withdrawal is successful.

testAaveBorrow 

**Tested key logic:** After 2 users do some operations (deposit/withdraw), if there is collateral in Aave, a user should be able to borrow.

**Tested user journey:**

- WalletOwner1: deposit some funds.
- WalletOwner2: withdraw some funds.
- WalletOwner2: deposit some funds.
- WalletOwner1: withdraw some funds.
- WalletOwner1: deposit some funds.
- WalletOwner1: borrow some funds.

**Test checks:**

- Check that the borrow is successful.

Contract: Cega

testCegaDeposit 

**Tested key logic:** After 2 users do some operations (deposit/withdraw), the wallet owner should still be able to deposit.

**Tested user journey:**

- WalletOwner1: deposit some funds.
- WalletOwner2: withdraw some funds.
- WalletOwner2: deposit some funds.
- WalletOwner1: withdraw some funds.
- WalletOwner1: deposit some funds.

**Test checks:**

- Check that the deposit is successful.

testCegaDepositLeverage 

**Tested key logic:** After 2 users do some operations (deposit/withdraw), the wallet owner should still be able to deposit with leverage.

**Tested user journey:**

- WalletOwner1: deposit some funds.
- WalletOwner2: withdraw some funds.
- WalletOwner2: deposit some funds.
- WalletOwner1: withdraw some funds.
- WalletOwner1: deposit some funds.
- WalletOwner1: withdraw some funds.



- WalletOwner1: deposit with leverage some funds.

**Test checks:**

- Check that the deposit with leverage is successful.

testCegaWithdrawal 

**Tested key logic:** After 2 users do some operations (deposit/withdraw), the wallet owner should still be able to withdraw if there are staked funds.

**Tested user journey:**

- WalletOwner1: deposit some funds.
- WalletOwner2: withdraw some funds.
- WalletOwner2: deposit some funds.
- WalletOwner1: withdraw some funds.
- WalletOwner1: deposit some funds.
- WalletOwner1: withdraw some funds.

**Test checks:**

- Check that the withdrawal is successful.

Contract: GearboxEarn

testGearboxEarnDeposit 

**Tested key logic:** After 2 users do some operations (deposit/withdraw), the wallet owner should still be able to deposit.

**Tested user journey:**

- WalletOwner1: deposit some funds.
- WalletOwner2: withdraw some funds.
- WalletOwner2: deposit some funds.
- WalletOwner1: withdraw some funds.
- WalletOwner1: deposit some funds.

**Test checks:**

- Check that the deposit is successful.

testGearboxEarnWithdraw 

**Tested key logic:** After 2 users do some operations (deposit/withdraw), the wallet owner should still be able to withdraw if there are staked funds.

**Tested user journey:**

- WalletOwner1: deposit some funds.
- WalletOwner2: withdraw some funds.
- WalletOwner2: deposit some funds.

- WalletOwner1: withdraw some funds.
- WalletOwner1: deposit some funds.
- WalletOwner1: withdraw some funds.

**Test checks:**

- Check that the withdrawal is successful.

Contract: InstaDapp

testInstaDappDeposit 

**Tested key logic:** After 2 users do some operations (deposit/withdraw), the wallet owner should still be able to deposit.

**Tested user journey:**

- WalletOwner1: deposit some funds.
- WalletOwner2: withdraw some funds.
- WalletOwner2: deposit some funds.
- WalletOwner1: withdraw some funds.
- WalletOwner1: deposit some funds.

**Test checks:**

- Check that the deposit is successful.

testInstaDappWithdraw 

**Tested key logic:** After 2 users do some operations (deposit/withdraw), the wallet owner should still be able to withdraw if there are staked funds.

**Tested user journey:**

- WalletOwner1: deposit some funds.
- WalletOwner2: withdraw some funds.
- WalletOwner2: deposit some funds.
- WalletOwner1: withdraw some funds.
- WalletOwner1: deposit some funds.
- WalletOwner1: withdraw some funds.

**Test checks:**

- Check that the withdrawal is successful.

Contract: Lido

testLidoDeposit 

**Tested key logic:** After 2 users do some operations (deposit/withdraw), the wallet owner should still be able to deposit.

**Tested user journey:**

- WalletOwner1: deposit some funds.
- WalletOwner2: withdraw some funds.
- WalletOwner2: deposit some funds.
- WalletOwner1: withdraw some funds.
- WalletOwner1: deposit some funds.

**Test checks:**

- Check that the deposit is successful.

testLidoWithdraw 

**Tested key logic:** After 2 users do some operations (deposit/withdraw), the wallet owner should still be able to withdraw if there are staked funds.

**Tested user journey:**

- WalletOwner1: deposit some funds.
- WalletOwner2: withdraw some funds.
- WalletOwner2: deposit some funds.
- WalletOwner1: withdraw some funds.
- WalletOwner1: deposit some funds.
- WalletOwner1: withdraw some funds.

**Test checks:**

- Check that the withdrawal is successful.

testLidoClaim 

**Tested key logic:** After 2 users do some operations (deposit/withdraw), the wallet owner should still be able to claim if the funds are ready to be claimed.

**Tested user journey:**

- WalletOwner1: deposit some funds.
- WalletOwner2: withdraw some funds.
- WalletOwner2: deposit some funds.
- WalletOwner1: withdraw some funds.
- WalletOwner1: deposit some funds.
- WalletOwner1: withdraw some funds.
- WalletOwner1: after some time, check if the funds are ready to be claimed and claim them.

**Test checks:**

- Check that the claim is successful if the funds are ready to be claimed.

testLidoWrap 

**Tested key logic:** After 2 users do some operations (deposit/withdraw), the wallet owner should still be able to wrap funds when possible.

**Tested user journey:**

- WalletOwner1: deposit some funds.
- WalletOwner2: withdraw some funds.
- WalletOwner2: deposit some funds.
- WalletOwner1: withdraw some funds.
- WalletOwner1: deposit some funds.
- WalletOwner1: wrap the funds.

**Test checks:**

- Check that the wrapping is successful.

Contract: RocketPool

testRocketPoolDeposit 

**Tested key logic:** After 2 users do some operations (deposit/withdraw), the wallet owner should still be able to deposit.

**Tested user journey:**

- WalletOwner1: deposit some funds.
- WalletOwner2: withdraw some funds.
- WalletOwner2: deposit some funds.
- WalletOwner1: withdraw some funds.
- WalletOwner1: deposit some funds.

**Test checks:**

- Check that the deposit is successful.

testRocketPoolWithdraw 

**Tested key logic:** After 2 users do some operations (deposit/withdraw), the wallet owner should still be able to withdraw if there are staked funds.

**Tested user journey:**

- WalletOwner1: deposit some funds.
- WalletOwner2: withdraw some funds.
- WalletOwner2: deposit some funds.
- WalletOwner1: withdraw some funds.
- WalletOwner1: deposit some funds.
- WalletOwner1: withdraw some funds.

**Test checks:**

- Check that the withdrawal is successful.

Contract: Unagi

testUnagiDeposit 

**Tested key logic:** After 2 users do some operations (deposit/withdraw), the wallet owner should still be able to deposit.

**Tested user journey:**

- WalletOwner1: deposit some funds.
- WalletOwner2: withdraw some funds.
- WalletOwner2: deposit some funds.
- WalletOwner1: withdraw some funds.
- WalletOwner1: deposit some funds.

**Test checks:**

- Check that the deposit is successful.

testUnagiWithdraw 

**Tested key logic:** After 2 users do some operations (deposit/withdraw), the wallet owner should still be able to withdraw if there are staked funds.

**Tested user journey:**

- WalletOwner1: deposit some funds.
- WalletOwner2: withdraw some funds.
- WalletOwner2: deposit some funds.
- WalletOwner1: withdraw some funds.
- WalletOwner1: deposit some funds.
- WalletOwner1: withdraw some funds.

**Test checks:**

- Check that the withdrawal is successful.

Contract: Vesper

testVesperDeposit 

**Tested key logic:** After 2 users do some operations (deposit/withdraw), the wallet owner should still be able to deposit.

**Tested user journey:**

- WalletOwner1: deposit some funds.
- WalletOwner2: withdraw some funds.
- WalletOwner2: deposit some funds.
- WalletOwner1: withdraw some funds.

- WalletOwner1: deposit some funds.

**Test checks:**

- Check that the deposit is successful.

testVesperWithdraw 

**Tested key logic:** After 2 users do some operations (deposit/withdraw), the wallet owner should still be able to withdraw if there are staked funds.

**Tested user journey:**

- WalletOwner1: deposit some funds.
- WalletOwner2: withdraw some funds.
- WalletOwner2: deposit some funds.
- WalletOwner1: withdraw some funds.
- WalletOwner1: deposit some funds.
- WalletOwner1: withdraw some funds.

**Test checks:**

- Check that the withdrawal is successful.

testVesperClaim 

**Tested key logic:** After 2 users do some operations (deposit/withdraw), the wallet owner should still be able to claim at any time.

**Tested user journey:**

- WalletOwner1: deposit some funds.
- WalletOwner1: attempt to claim.
- WalletOwner2: withdraw some funds.
- WalletOwner2: attempt to claim.
- WalletOwner2: deposit some funds.
- WalletOwner2: attempt to claim.
- WalletOwner1: withdraw some funds.
- WalletOwner1: attempt to claim.
- WalletOwner1: deposit some funds.
- WalletOwner1: attempt to claim.
- WalletOwner1: withdraw some funds.
- WalletOwner1: attempt to claim.

**Test checks:**

- Check that the claim does not revert.

Contract: VesperGovernance

testVesperGovernanceDeposit 

**Tested key logic:** After 2 users do some operations (deposit/withdraw), the wallet owner should still be able to deposit.

**Tested user journey:**

- WalletOwner1: deposit some funds.
- WalletOwner2: withdraw some funds.
- WalletOwner2: deposit some funds.
- WalletOwner1: withdraw some funds.
- WalletOwner1: deposit some funds.

**Test checks:**

- Check that the deposit is successful.

testVesperGovernanceWithdraw 

**Tested key logic:** After 2 users do some operations (deposit/withdraw), the wallet owner should still be able to withdraw if there are staked funds.

**Tested user journey:**

- WalletOwner1: deposit some funds.
- WalletOwner2: withdraw some funds.
- WalletOwner2: deposit some funds.
- WalletOwner1: withdraw some funds.
- WalletOwner1: deposit some funds.
- WalletOwner1: withdraw some funds.

**Test checks:**

- Check that the withdrawal is successful.

Contract: UniswapV2

testUniswapV2Deposit 

**Tested key logic:** After 2 users do some operations (deposit/withdraw), the wallet owner should still be able to deposit.

**Tested user journey:**

- WalletOwner1: deposit some funds.
- WalletOwner2: withdraw some funds.
- WalletOwner2: deposit some funds.
- WalletOwner1: withdraw some funds.
- WalletOwner1: deposit some funds.

**Test checks:**

- Check that the deposit is successful.

testUniswapV2Withdraw 

**Tested key logic:** After 2 users do some operations (deposit/withdraw), the wallet owner should still be able to withdraw if there are staked funds.

**Tested user journey:**

- WalletOwner1: deposit some funds.
- WalletOwner2: withdraw some funds.
- WalletOwner2: deposit some funds.
- WalletOwner1: withdraw some funds.
- WalletOwner1: deposit some funds.
- WalletOwner1: withdraw some funds.

**Test checks:**

- Check that the withdrawal is successful.

Contract: UniswapV3

testUniswapV3Deposit 

**Tested key logic:** After 2 users do some operations (deposit/withdraw), the wallet owner should still be able to deposit.

**Tested user journey:**

- WalletOwner1: deposit some funds.
- WalletOwner2: withdraw some funds.
- WalletOwner2: deposit some funds.
- WalletOwner1: withdraw some funds.
- WalletOwner1: deposit some funds.

**Test checks:**

- Check that the deposit is successful.

testUniswapV3Withdraw 

**Tested key logic:** After 2 users do some operations (deposit/withdraw), the wallet owner should still be able to withdraw if there are staked funds.

**Tested user journey:**

- WalletOwner1: deposit some funds.
- WalletOwner2: deposit some funds.
- WalletOwner1: deposit some funds.
- WalletOwner1: withdraw some funds.



**Test checks:**

- Check that the withdrawal is successful.

testUniswapV3Claim 

**Tested key logic:** After 2 users do some operations (deposit/withdraw), the wallet owner should still be able to claim.

**Tested user journey:**

- WalletOwner1: deposit some funds.
- WalletOwner2: withdraw some funds.
- WalletOwner2: deposit some funds.
- WalletOwner1: withdraw some funds.
- WalletOwner1: deposit some funds.
- WalletOwner1: withdraw some funds.
- WalletOwner1: claim.

**Test checks:**

- Check that the claim is successful.

testUniswapV3WithdrawNFT 

**Tested key logic:** After 2 users do some operations (deposit/withdraw), the wallet owner should still be able to withdraw the NFT if there are staked funds.

**Tested user journey:**

- WalletOwner1: deposit some funds.
- WalletOwner2: withdraw some funds.
- WalletOwner2: deposit some funds.
- WalletOwner1: withdraw some funds.
- WalletOwner1: deposit some funds.
- WalletOwner1: deposit some funds.
- WalletOwner1: withdraw the NFT.

**Test checks:**

- Check that the withdrawal is successful.

testUniswapV3UpdatePosition 

**Tested key logic:** After 2 users do some operations (deposit/withdraw), the wallet owner should still be able to update an existing position.

**Tested user journey:**

- WalletOwner1: deposit some funds.
- WalletOwner2: deposit some funds.
- WalletOwner1: withdraw some funds.
- WalletOwner1: deposit some funds.

- WalletOwner1: update the position.

**Test checks:**

- Check that the position update is successful.

Contract: NotionalLP

testNotionalLPDeposit 

**Tested key logic:** After 2 users do some operations (deposit/withdraw), the wallet owner should still be able to deposit.

**Tested user journey:**

- WalletOwner1: deposit some funds.
- WalletOwner1: withdraw some funds.
- WalletOwner2: deposit some funds.
- WalletOwner2: withdraw some funds.
- WalletOwner1: deposit some funds.

**Test checks:**

- Check that the deposit is successful.

testNotionalLPWithdraw 

**Tested key logic:** After 2 users do some operations (deposit/withdraw), the wallet owner should still be able to withdraw if there are staked funds.

**Tested user journey:**

- WalletOwner1: deposit some funds.
- WalletOwner1: withdraw some funds.
- WalletOwner2: deposit some funds.
- WalletOwner2: withdraw some funds.
- WalletOwner1: deposit some funds.
- WalletOwner2: withdraw some funds.

**Test checks:**

- Check that the withdrawal is successful.

testNotionalLPClaim 

**Tested key logic:** After 2 users do some operations (deposit/withdraw), the wallet owner should still be able to claim.

**Tested user journey:**

- WalletOwner1: deposit some funds.
- WalletOwner1: claim after some time.
- WalletOwner1: withdraw some funds.

- WalletOwner2: deposit some funds.
- WalletOwner2: claim after some time.
- WalletOwner1: deposit some funds.
- WalletOwner2: withdraw some funds after some time.
- WalletOwner1: claim.

**Test checks:**

- Check that the claim is successful.

Contract: NotionalFD

testNotionalFDDeposit 

**Tested key logic:** After 2 users do some operations (deposit/withdraw), the wallet owner should still be able to deposit.

**Tested user journey:**

- WalletOwner1: deposit some funds.
- WalletOwner2: deposit some funds.
- WalletOwner2: withdraw some funds.
- WalletOwner1: withdraw some funds.
- WalletOwner1: deposit some funds.

**Test checks:**

- Check that the deposit is successful.

testNotionalFDWithdraw 

**Tested key logic:** After 2 users do some operations (deposit/withdraw), the wallet owner should still be able to withdraw if there are staked funds.

**Tested user journey:**


- WalletOwner1: deposit some funds.
- WalletOwner2: deposit some funds.
- WalletOwner2: withdraw some funds.
- WalletOwner1: withdraw some funds.
- WalletOwner1: deposit some funds.
- WalletOwner1: withdraw some funds after a certain amount of time.

**Test checks:**

- Check that the withdrawal is successful.

## Invariant Tests

Contract: Diamond

invariantDiamondOwner 

**Tested key logic:** Ownership can only be modified by the current owner.

### Invariant setup:

- Owner: set an admin.
- Owner: set the protocol fee collector and multiplier.
- Owner: set the flash loan handler.
- Owner: set the NFT handler.
- Owner: set some blacklist and whitelist.
- Create 2 user wallets.

### Invariant checks:

- Check that the owner remains the same.

Contract: Admin

invariantDiamondAdmin 

**Tested key logic:** An admin can only be set by the diamond owner.

### Invariant setup:

- Owner: set an admin.
- Owner: set the protocol fee collector and multiplier.
- Owner: set the flash loan handler.
- Owner: set the NFT handler.
- Owner: set some blacklist and whitelist.
- Create 2 user wallets.

### Invariant checks:

- Check that the admin remains the same.

Contract: ProtocolFee

invariantFeeConfig 

**Tested key logic:** The Fee collector and the Fee multiplier can only be set by the owner.

### Invariant setup:

- Owner: set an admin.
- Owner: set the protocol fee collector and multiplier.
- Owner: set the flash loan handler.

- Owner: set the NFT handler.
- Owner: set some blacklist and whitelist.
- Create 2 user wallets.

**Invariant checks:**

- Check that the fee collector and fee multiplier remain the same.

Contract: FlashLoanHandler

invariantFlashloanHandlerCallerAndInput 

**Tested key logic:** The authorized caller and the handler execution can only be changed by the diamond owner.

**Invariant setup:**

- Owner: set an admin.
- Owner: set the protocol fee collector and multiplier.
- Owner: set the flash loan handler.
- Owner: set the NFT handler.
- Owner: set some blacklist and whitelist.
- Create 2 user wallets.

**Invariant checks:**

- Check that the fee collector and fee multiplier remain the same.

Contract: Nft

invariantNFTHandler 

**Tested key logic:** Only the diamond owner can set the NFT handler.

**Invariant setup:**

- Owner: set an admin.
- Owner: set the protocol fee collector and multiplier.
- Owner: set the flash loan handler.
- Owner: set the NFT handler.
- Owner: set some blacklist and whitelist.
- Create 2 user wallets.

**Invariant checks:**

- Check that the flash loan handler and caller remain the same.

Contract: SmartWalletAuth

invariantWhitelistFn 

**Tested key logic:** Only the diamond owner can set whitelisted functions.

**Invariant setup:**

- Owner: set an admin.
- Owner: set the protocol fee collector and multiplier.
- Owner: set the flash loan handler.
- Owner: set the NFT handler.
- Owner: set some blacklist and whitelist.
- Create 2 user wallets.

**Invariant checks:**

- Check that the functions were correctly whitelisted.

invariantBlacklistFn 

**Tested key logic:** Only the diamond owner can set blacklisted functions.

**Invariant setup:**

- Owner: set an admin.
- Owner: set the protocol fee collector and multiplier.
- Owner: set the flash loan handler.
- Owner: set the NFT handler.
- Owner: set some blacklist and whitelist.
- Create 2 user wallets.

**Invariant checks:**

- Check that the functions were correctly blacklisted.

Contract: FeeCollector

invariantOwner 

**Tested key logic:** Ownership can only change if the current owner requested it.

**Invariant setup:**

- Owner: deploy the fee collector.

**Invariant checks:**

- Check that the owner remains unchanged.

invariantOwnerClaim 

**Tested key logic:** The owner can claim the fees.

**Invariant setup:**

- Owner: deploy the fee collector.

**Invariant checks:**

- Check that the owner claims the fees successfully.

invariantUserClaim 

**Tested key logic:** Referrers can claim their referral fees.

**Invariant setup:**

- Owner: deploy the fee collector.

**Invariant checks:**

- Check that the referrers claim their fees successfully.

Contract: AaveV3

invariantDepositWithdraw 

**Tested key logic:** A user that deposits in AaveV3 can withdraw at any time.

**Invariant setup:**

- Owner: whitelist the necessary functions.
- Create 2 user wallets and give them some funds.
- WalletOwner1: deposit some funds.

**Invariant checks:**

- Check that WalletOwner1 can withdraw his deposit.

invariantUserProfitRangeBroken 

**Tested key logic:** Under a reasonable time limit a user cannot end up with excessive returns in AaveV3.

**Invariant setup:**

- Owner: whitelist the necessary functions.
- Create 2 user wallets and give them some funds.
- WalletOwner1: deposit some funds.

**Invariant checks:**

- Check that the returns are not excessive which might suggest funds theft.

Contract: Cega

invariantDepositWithdraw 

**Tested key logic:** A user that deposits in Cega can withdraw at any time.

**Invariant setup:**

- Owner: whitelist the necessary functions.
- Create 2 user wallets and give them some funds.

- WalletOwner1: deposit some funds.

**Invariant checks:**

- Check that WalletOwner1 can withdraw his deposit.

invariantUserProfitRangeBroken 

**Tested key logic:** Under a reasonable time limit a user cannot end up with excessive returns in Cega.

**Invariant setup:**

- Owner: whitelist the necessary functions.
- Create 2 user wallets and give them some funds.
- WalletOwner1: deposit some funds.

**Invariant checks:**

- Check that the returns are not excessive which might suggest funds theft.

Contract: GearboxEarn

invariantDepositWithdraw 

**Tested key logic:** A user that deposits in GearboxEarn can withdraw at any time.

**Invariant setup:**

- Owner: whitelist the necessary functions.
- Create 2 user wallets and give them some funds.
- WalletOwner1: deposit some funds.

**Invariant checks:**

- Check that WalletOwner1 can withdraw his deposit.

invariantUserProfitRangeBroken 

**Tested key logic:** Under a reasonable time limit a user cannot end up with excessive returns in GearboxEarn.

**Invariant setup:**

- Owner: whitelist the necessary functions.
- Create 2 user wallets and give them some funds.
- WalletOwner1: deposit some funds.

**Invariant checks:**

- Check that the returns are not excessive which might suggest funds theft.



Contract: InstaDapp

invariantDepositWithdraw 

**Tested key logic:** A user that deposits in InstaDapp can withdraw at any time.

**Invariant setup:**

- Owner: whitelist the necessary functions.
- Create 2 user wallets and give them some funds.
- WalletOwner1: deposit some funds.

**Invariant checks:**

- Check that WalletOwner1 can withdraw his deposit.

invariantUserProfitRangeBroken 

**Tested key logic:** Under a reasonable time limit a user cannot end up with excessive returns in InstaDapp.

**Invariant setup:**

- Owner: whitelist the necessary functions.
- Create 2 user wallets and give them some funds.
- WalletOwner1: deposit some funds.

**Invariant checks:**

- Check that the returns are not excessive which might suggest funds theft.

Contract: Lido

invariantDepositWithdraw 

**Tested key logic:** A user that deposits in Lido can withdraw at any time.

**Invariant setup:**

- Owner: whitelist the necessary functions.
- Create 2 user wallets and give them some funds.
- WalletOwner1: deposit some funds.

**Invariant checks:**

- Check that WalletOwner1 can withdraw his deposit.

invariantUserProfitRangeBroken 

**Tested key logic:** Under a reasonable time limit a user cannot end up with excessive returns in Lido.

**Invariant setup:**

- Owner: whitelist the necessary functions.

- Create 2 user wallets and give them some funds.
- WalletOwner1: deposit some funds.

**Invariant checks:**

- Check that the returns are not excessive which might suggest funds theft.

Contract: NotionalFD

invariantDepositWithdraw 

**Tested key logic:** A user that deposits in NotionalFD can withdraw at any time.

**Invariant setup:**

- Owner: whitelist the necessary functions.
- Create 2 user wallets and give them some funds.
- WalletOwner1: deposit some funds.

**Invariant checks:**

- Check that WalletOwner1 can withdraw his deposit.

invariantUserProfitRangeBroken 

**Tested key logic:** Under a reasonable time limit a user cannot end up with excessive returns in NotionalFD.

**Invariant setup:**

- Owner: whitelist the necessary functions.
- Create 2 user wallets and give them some funds.
- WalletOwner1: deposit some funds.

**Invariant checks:**

- Check that the returns are not excessive which might suggest funds theft.

Contract: NotionalLP

invariantDepositWithdraw 

**Tested key logic:** A user that deposits in NotionalLP can withdraw at any time.

**Invariant setup:**

- Owner: whitelist the necessary functions.
- Create 2 user wallets and give them some funds.
- WalletOwner1: deposit some funds.

**Invariant checks:**

- Check that WalletOwner1 can withdraw his deposit.

invariantUserProfitRangeBroken 

**Tested key logic:** Under a reasonable time limit a user cannot end up with excessive returns in NotionalLP.

**Invariant setup:**

- Owner: whitelist the necessary functions.
- Create 2 user wallets and give them some funds.
- WalletOwner1: deposit some funds.

**Invariant checks:**

- Check that the returns are not excessive which might suggest funds theft.

Contract: RocketPool

invariantDepositWithdraw 

**Tested key logic:** A user that deposits in RocketPool can withdraw at any time.

**Invariant setup:**

- Owner: whitelist the necessary functions.
- Create 2 user wallets and give them some funds.
- WalletOwner1: deposit some funds.

**Invariant checks:**

- Check that WalletOwner1 can withdraw his deposit.

invariantUserProfitRangeBroken 

**Tested key logic:** Under a reasonable time limit a user cannot end up with excessive returns in RocketPool.

**Invariant setup:**

- Owner: whitelist the necessary functions.
- Create 2 user wallets and give them some funds.
- WalletOwner1: deposit some funds.

**Invariant checks:**

- Check that the returns are not excessive which might suggest funds theft.

Contract: Unagii

invariantDepositWithdraw 

**Tested key logic:** A user that deposits in Unagii can withdraw at any time.

**Invariant setup:**

- Owner: whitelist the necessary functions.
- Create 2 user wallets and give them some funds.
- WalletOwner1: deposit some funds.

**Invariant checks:**

- Check that WalletOwner1 can withdraw his deposit.

invariantUserProfitRangeBroken 

**Tested key logic:** Under a reasonable time limit a user cannot end up with excessive returns in Unagii.

**Invariant setup:**

- Owner: whitelist the necessary functions.
- Create 2 user wallets and give them some funds.
- WalletOwner1: deposit some funds.

**Invariant checks:**

- Check that the returns are not excessive which might suggest funds theft.

Contract: UniswapV2

invariantDepositWithdraw 

**Tested key logic:** A user that deposits in UniswapV2 can withdraw at any time.

**Invariant setup:**

- Owner: whitelist the necessary functions.
- Create 2 user wallets and give them some funds.
- WalletOwner1: deposit some funds.

**Invariant checks:**

- Check that WalletOwner1 can withdraw his deposit.

invariantUserProfitRangeBroken 

**Tested key logic:** Under a reasonable time limit a user cannot end up with excessive returns in UniswapV2.

**Invariant setup:**

- Owner: whitelist the necessary functions.
- Create 2 user wallets and give them some funds.
- WalletOwner1: deposit some funds.

**Invariant checks:**

- Check that the returns are not excessive which might suggest funds theft.

Contract: UniswapV3

invariantDepositWithdraw 

**Tested key logic:** A user that deposits in UniswapV3 can withdraw at any time.

**Invariant setup:**

- Owner: whitelist the necessary functions.
- Create 2 user wallets and give them some funds.
- WalletOwner1: deposit some funds.

**Invariant checks:**

- Check that WalletOwner1 can withdraw his deposit.

invariantUserProfitRangeBroken 

**Tested key logic:** Under a reasonable time limit a user cannot end up with excessive returns in UniswapV3.

**Invariant setup:**

- Owner: whitelist the necessary functions.
- Create 2 user wallets and give them some funds.
- WalletOwner1: deposit some funds.

**Invariant checks:**

- Check that the returns are not excessive which might suggest funds theft.

Contract: Vesper

invariantDepositWithdraw 

**Tested key logic:** A user that deposits in Vesper can withdraw at any time.

**Invariant setup:**

- Owner: whitelist the necessary functions.
- Create 2 user wallets and give them some funds.
- WalletOwner1: deposit some funds.

**Invariant checks:**

- Check that WalletOwner1 can withdraw his deposit.

invariantUserProfitRangeBroken 

**Tested key logic:** Under a reasonable time limit a user cannot end up with excessive returns in Vesper.

**Invariant setup:**

- Owner: whitelist the necessary functions.

- Create 2 user wallets and give them some funds.
- WalletOwner1: deposit some funds.

**Invariant checks:**

- Check that the returns are not excessive which might suggest funds theft.

Contract: VesperGovernance

invariantDepositWithdraw 

**Tested key logic:** A user that deposits in VesperGovernance can withdraw at any time.

**Invariant setup:**

- Owner: whitelist the necessary functions.
- Create 2 user wallets and give them some funds.
- WalletOwner1: deposit some funds.

**Invariant checks:**

- Check that WalletOwner1 can withdraw his deposit.

invariantUserProfitRangeBroken 

**Tested key logic:** Under a reasonable time limit a user cannot end up with excessive returns in VesperGovernance.

**Invariant setup:**

- Owner: whitelist the necessary functions.
- Create 2 user wallets and give them some funds.
- WalletOwner1: deposit some funds.

**Invariant checks:**

- Check that the returns are not excessive which might suggest funds theft.

Contract: DezyDCA

invariantDepositWithdraw 

**Tested key logic:** A user that deposits in DezyDCA can withdraw at any time.

**Invariant setup:**

- Owner: whitelist the necessary functions.
- Create 2 user wallets and give them some funds.
- WalletOwner1: deposit some funds.

**Invariant checks:**

- Check that WalletOwner1 can withdraw his deposit.

invariantUserProfitRangeBroken 

**Tested key logic:** Under a reasonable time limit a user cannot end up with excessive returns in DezyDCA.

**Invariant setup:**

- Owner: whitelist the necessary functions.
- Create 2 user wallets and give them some funds.
- WalletOwner1: deposit some funds.

**Invariant checks:**

- Check that the returns are not excessive which might suggest funds theft.

invariantFees 

**Tested key logic:** Check that only the owner can set fees.

**Invariant setup:**

- Owner: deploy a DCAStrategy contract.
- Give funds to 2 users: user1 and user2.

**Invariant checks:**

- Check that the fees remain unchanged.

invariantPause 

**Tested key logic:** Check that only the owner can switch the pause/unpause feature.

**Invariant setup:**

- Owner: deploy a DCAStrategy contract.
- Give funds to 2 users: user1 and user2.

**Invariant checks:**

- Check that the pause status remains unchanged.

Contract: Fees

invariantFees 

**Tested key logic:** Only the owner can set the token fees.

**Invariant setup:**

- Owner: deploy the Fees contract.
- Owner: set the token fee, deposit status, fee collector, and whitelisted tokens.

**Invariant checks:**

- Check that the token fee settings remain unchanged.

invariantDepositStatus 

**Tested key logic:** Only the owner can set the deposit status.

**Invariant setup:**

- Owner: deploy the Fees contract.
- Owner: set the token fee, deposit status, fee collector, and whitelisted tokens.

**Invariant checks:**

- Check that the deposit status remains unchanged.

invariantFeeCollector 

**Tested key logic:** Only the owner can set the fee collector.

**Invariant setup:**

- Owner: deploy the Fees contract.
- Owner: set the token fee, deposit status, fee collector, and whitelisted tokens.

**Invariant checks:**

- Check that the fee collector remains unchanged.

invariantDefaultFee 

**Tested key logic:** Only the owner can set the default fee.

**Invariant setup:**

- Owner: deploy the Fees contract.
- Owner: set the token fee, deposit status, fee collector, and whitelisted tokens.

**Invariant checks:**

- Check that the default fee remains unchanged.

invariantWhitelistTokens 

**Tested key logic:** Only the owner can whitelist tokens.

**Invariant setup:**

- Owner: deploy the Fees contract.
- Owner: set the token fee, deposit status, fee collector, and whitelisted tokens.

**Invariant checks:**

- Check that the whitelisted token is still whitelisted.



Contract: OracleAggregator

invariantChainlinkTokens 

**Tested key logic:** Only the owner can set the Chainlink tokens.

**Invariant setup:**

- Owner: deploy the OracleAggregator contract and initialize it.
- Owner: set the Chainlink tokens and the Uniswap pools.

**Invariant checks:**

- Check that the chainlink token hasn't changed.

invariantUniswapPools 

**Tested key logic:** Only the owner can set the Uniswap pools.

**Invariant setup:**

- Owner: deploy the OracleAggregator contract and initialize it.
- Owner: set the Chainlink tokens and the Uniswap pools.

**Invariant checks:**

- Check that the Uniswap pool hasn't changed.

# Relevant Past Hacks

We have identified the following publicly known past hacks that are relevant to the DezyV2 project which is similar in functionality to yield farming in general.

## Address misconfiguration

**Threat Model:** A wrong address configuration causes a wrong price calculation.

Yearn Finance

**Link:** [https://github.com/SunWeb3Sec/DeFiHackLabs/blob/main/src/test/YearnFinance\\_exp.sol](https://github.com/SunWeb3Sec/DeFiHackLabs/blob/main/src/test/YearnFinance_exp.sol)

### Code Review

Addresses that are hardcoded are correct, and the DezyStrategy uses a price oracle to derive prices.

## Arbitrary unstake

**Threat Model:** Arbitrary unstake function allowed the attacker to unstake users' funds and get them.

Thena

**Link:** [https://github.com/SunWeb3Sec/DeFiHackLabs/blob/main/src/test/Thena\\_exp.sol](https://github.com/SunWeb3Sec/DeFiHackLabs/blob/main/src/test/Thena_exp.sol)

### Code Review

The wallet requires authentication before unstaking can happen for any strategy.

## Wrong accounting

**Threat Model:** Wrong account when considering an EOA and a contract.

Spherax USDs

**Link:** [https://github.com/SunWeb3Sec/DeFiHackLabs/blob/main/src/test/USDs\\_exp.sol](https://github.com/SunWeb3Sec/DeFiHackLabs/blob/main/src/test/USDs_exp.sol)

### Code Review

The wallet diamond is the entity that directly interacts with strategies. There is no accounting distinction between EOA and contracts as wallet owners.

## Custom ERC20 implementation

**Threat Model:** The custom implementation was increasing the attacker's balance on each transfer.

Thorem Finance

**Link:**

[https://github.com/SunWeb3Sec/DeFiHackLabs/blob/main/src/test/ThoremFinance\\_exp.sol](https://github.com/SunWeb3Sec/DeFiHackLabs/blob/main/src/test/ThoremFinance_exp.sol)

**Code Review** ✓

Dezy does not have a custom ERC20 implementation.

## The calculation of rewards depended on the caller's balance

**Threat Model:** There is no time factor when calculating the rewards. The attacker exploited it by using a flash loan to amplify the rewards.

NewFreeDAO

**Link:** [https://github.com/SunWeb3Sec/DeFiHackLabs/blob/main/src/test/NewFreeDAO\\_exp.sol](https://github.com/SunWeb3Sec/DeFiHackLabs/blob/main/src/test/NewFreeDAO_exp.sol)

**Code Review** ✓

The wallet diamond is the entity that directly interacts with strategies. There is no accounting distinction between EOA and contracts as wallet owners.

## Reentrancy

**Threat Model:** The lack of input sanitization allowed a wrong LP to be referenced plus the ability to have a callback into the attacker-controlled tokens allowed the attacker to drain funds.

Paraluni

**Link:** [https://github.com/SunWeb3Sec/DeFiHackLabs/blob/main/src/test/Paraluni\\_exp.sol](https://github.com/SunWeb3Sec/DeFiHackLabs/blob/main/src/test/Paraluni_exp.sol)

**Invariant tested** ✓

InvariantProfitRangeBroken

We have identified the following publicly known past hacks that are relevant to the OracleAggregator contract which is similar in functionality to a price oracle in general.

## Price Oracle manipulation

**Threat Model:** Manipulation of the price oracle either via pool reserves manipulation or by donation.

RodeoFinance

**Link:** [https://github.com/SunWeb3Sec/DeFiHackLabs/blob/main/src/test/RodeoFinance\\_exp.sol](https://github.com/SunWeb3Sec/DeFiHackLabs/blob/main/src/test/RodeoFinance_exp.sol)

0vix

**Link:** [https://github.com/SunWeb3Sec/DeFiHackLabs/blob/main/src/test/0vix\\_exp.sol](https://github.com/SunWeb3Sec/DeFiHackLabs/blob/main/src/test/0vix_exp.sol)

BonqDAO

**Link:** [https://github.com/SunWeb3Sec/DeFiHackLabs/blob/main/src/test/BonqDAO\\_exp.sol](https://github.com/SunWeb3Sec/DeFiHackLabs/blob/main/src/test/BonqDAO_exp.sol)

Inverse Finance

**Link:** [https://github.com/SunWeb3Sec/DeFiHackLabs/blob/main/src/test/InverseFinance\\_exp.sol](https://github.com/SunWeb3Sec/DeFiHackLabs/blob/main/src/test/InverseFinance_exp.sol)

### Code Review

Dezy uses Chainlink and Uniswap V3 as a fallback. The past hacks should not be applicable. The price oracle in the DCA strategy can only be initialized once.

## Public price setting

**Threat Model:** Public function to set the price.

Fortress Loans

**Link:** <https://github.com/SunWeb3Sec/DeFiHackLabs/blob/main/src/test/FortressLoans.exp.sol>

### Code Review

Dezy uses Chainlink and Uniswap V3 as a fallback. The past hacks should not be applicable. The price oracle in the DCA strategy can only be initialized once.

## Public set price oracle

**Threat Model:** Public function to set the price oracle.

Rikkei Finance

**Link:** [https://github.com/SunWeb3Sec/DeFiHackLabs/blob/main/src/test/Rikkei\\_exp.sol](https://github.com/SunWeb3Sec/DeFiHackLabs/blob/main/src/test/Rikkei_exp.sol)

### **Code Review**

Dezy uses Chainlink and Uniswap V3 as a fallback. The past hacks should not be applicable. The price oracle in the DCA strategy can only be initialized once.

# Fix Log

**Table 2** Fix Log

Title	Severity	Status
<a href="#">1. No slippage protection in UniswapV2</a>	Medium	Fixed at commit 7c8634b8fca1288ba1c94 444cf513c2bed6aeba0
<a href="#">2. The swap deadline is not correctly set</a>	Medium	Fixed at commit 7c8634b8fca1288ba1c94 444cf513c2bed6aeba0
<a href="#">3. The aaveV3 module is vulnerable to potential sandwich attack</a>	Medium	Fixed at commit 3e9b4937af925d4b5687 d2727562a2a1b8555da9
<a href="#">4. Missing LibuniswapV3Mapping update when calling decreaseLiquidity()</a>	Medium	Fixed at commit d807323a33a92024db26 1b7d3528f9500448c5c0
<a href="#">5. Incorrect index update in LibSmartWalletAuth</a>	Medium	Fixed at commit a8406ab8b6b14081e9ed f8c36349d22193c8a2fc
<a href="#">6. Potential reentrancy in SmartWalletAA</a>	Low	Fixed at commit 29000834856663b01750 072d9a043a966a444295
<a href="#">7. Incorrect funcSig value in SmartWalletMulticall</a>	Low	Fixed at commit 200160870c2efb3951e2d e98135398ca8b6b07d4
<a href="#">8. Unmatched docstring in LibReentrancyGuard</a>	Informational	Fixed at commit af282e479ac534923f007 7d97f32baa4dd093599
<a href="#">9. Zero should not be used as the magic value to represent the entire balance</a>	Informational	Fixed at commit 24c432e1a46767b318c5 67d187b3499b9e3e9733

# Appendix

## Severity Categories

Severity	Description
<b>Gas</b>	Gas optimization.
<b>Informational</b>	The issue does not pose a security risk but is relevant to best security practices.
<b>Low</b>	The issue does not put assets at risk such as functions being inconsistent with specifications or issues with comments.
<b>Medium</b>	The issue puts assets at risk not directly but with a hypothetical attack path, stated assumptions, or external requirements. Or the issue impacts the functionalities or availability of the protocol.
<b>High</b>	The issue directly results in assets being stolen, lost, or compromised.