// Smart Contract Security Assessment 12.26.2024 - 01.15.2025

Unleash Smart Contracts

Unleash

HALBORN

Unleash Smart Contracts - Unleash

Prepared by: HALBORN

Last Updated 02/10/2025

Date of Engagement by: December 26th, 2024 - January 15th, 2025

Summary

100% ⓘ OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
10	0	2	2	2	4

TABLE OF CONTENTS

1. Introduction
2. Assessment summary
3. Test approach and methodology
4. Risk methodology
5. Scope
6. Assessment summary & findings overview
7. Findings & Tech Details
    - 7.1 Malicious delegatees can block all delegators funds
    - 7.2 Initial proposal passes with minimal stake
    - 7.3 Confidence intervals from pyth oracle are ignored
    - 7.4 Missing zero reserves check in uniswapv2library
    - 7.5 Unrestricted pyth price updates
    - 7.6 Incorrect cei pattern implementation in stakelib's unstake()
    - 7.7 Use of unlicensed smart contracts
    - 7.8 Use of custom errors instead of revert strings
    - 7.9 Integration with undocumented protocol
    - 7.10 Unused imports
8. Automated Testing

1. Introduction

Unleash Protocol engaged Halborn to conduct a security assessment on their smart contracts revisions started on December 26th, 2024 and ending on January 15th, 2025. The security assessment was scoped to the smart contracts provided to the Halborn team.

Commit hashes and further details can be found in the Scope section of this report.

2. Assessment Summary

The team at Halborn assigned a security engineer to evaluate the security of the smart contract. The security engineer is a blockchain and smart-contract security expert with advanced penetration testing, smart-contract hacking, and deep knowledge of multiple blockchain protocols.

The purpose of this assessment is to:

- Ensure that smart contract functions operate as intended.
- Identify potential security issues with the smart contracts.

In summary, Halborn identified some improvements to reduce the likelihood and impact of risks, which have been partially addressed by the Unleash team. The main ones were the following:

- Prevent users from delegating to the zero address.
- Record a token's price in AaveOracle contract, the first time it is accessed in each block.
- Incorporate checks on the confidence interval to ensure that the price is reliable before using it.
- Add reserve validation checks in both uniswapV2 functions.

3. Test Approach And Methodology

Halborn performed a combination of manual and automated security testing to balance efficiency, timeliness, practicality, and accuracy regarding the scope of this assessment. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance code coverage and quickly identify items that do not follow the security best practices. The following phases and associated tools were used during the assessment:

- Research into architecture and purpose.
- Smart contract manual code review and walkthrough.
- Graphing out functionality and contract logic/connectivity/functions. (`solgraph`,`draw.io`)
- Manual assessment of use and safety for the critical Solidity variables and functions in scope to identify any arithmetic related vulnerability classes.
- Manual testing by custom scripts.
- Static Analysis of security for scoped contract, and imported functions. (`Slither`)
- Testnet deployment. (`Hardhat`,`Foundry`)

4. RISK METHODOLOGY

Every vulnerability and issue observed by Halborn is ranked based on **two sets** of **Metrics** and a **Severity Coefficient**. This system is inspired by the industry standard Common Vulnerability Scoring System.

The two **Metric sets** are: **Exploitability** and **Impact**. **Exploitability** captures the ease and technical means by which vulnerabilities can be exploited and **Impact** describes the consequences of a successful exploit.

The **Severity Coefficients** is designed to further refine the accuracy of the ranking with two factors: **Reversibility** and **Scope**. These capture the impact of the vulnerability on the environment as well as the number of users and smart contracts affected.

The final score is a value between 0-10 rounded up to 1 decimal place and 10 corresponding to the highest security risk. This provides an objective and accurate rating of the severity of security vulnerabilities in smart contracts.

The system is designed to assist in identifying and prioritizing vulnerabilities based on their level of risk to address the most critical issues in a timely manner.

4.1 EXPLOITABILITY

ATTACK ORIGIN (AO):

Captures whether the attack requires compromising a specific account.

ATTACK COST (AC):

Captures the cost of exploiting the vulnerability incurred by the attacker relative to sending a single transaction on the relevant blockchain. Includes but is not limited to financial and computational cost.

ATTACK COMPLEXITY (AX):

Describes the conditions beyond the attacker's control that must exist in order to exploit the vulnerability. Includes but is not limited to macro situation, available third-party liquidity and regulatory challenges.

METRICS:

EXPLOITABILITY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Origin (AO)	Arbitrary (AO:A) Specific (AO:S)	1 0.2
Attack Cost (AC)	Low (AC:L) Medium (AC:M) High (AC:H)	1 0.67 0.33

EXPLOITABILITY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Complexity (AX)	Low (AX:L) Medium (AX:M) High (AX:H)	1 0.67 0.33

Exploitability E is calculated using the following formula:

$$E = \prod m_e$$

4.2 IMPACT

CONFIDENTIALITY (C):

Measures the impact to the confidentiality of the information resources managed by the contract due to a successfully exploited vulnerability. Confidentiality refers to limiting access to authorized users only.

INTEGRITY (I):

Measures the impact to integrity of a successfully exploited vulnerability. Integrity refers to the trustworthiness and veracity of data stored and/or processed on-chain. Integrity impact directly affecting Deposit or Yield records is excluded.

AVAILABILITY (A):

Measures the impact to the availability of the impacted component resulting from a successfully exploited vulnerability. This metric refers to smart contract features and functionality, not state. Availability impact directly affecting Deposit or Yield is excluded.

DEPOSIT (D):

Measures the impact to the deposits made to the contract by either users or owners.

YIELD (Y):

Measures the impact to the yield generated by the contract for either users or owners.

METRICS:

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
Confidentiality (C)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
Integrity (I)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1
Availability (A)	None (A:N) Low (A:L) Medium (A:M) High (A:H) Critical (A:C)	0 0.25 0.5 0.75 1
Deposit (D)	None (D:N) Low (D:L) Medium (D:M) High (D:H) Critical (D:C)	0 0.25 0.5 0.75 1
Yield (Y)	None (Y:N) Low (Y:L) Medium (Y:M) High (Y:H) Critical (Y:C)	0 0.25 0.5 0.75 1

Impact I is calculated using the following formula:

$$I = max(m_I) + \frac{\sum m_I - max(m_I)}{4}$$

4.3 SEVERITY COEFFICIENT

REVERSIBILITY (R):

Describes the share of the exploited vulnerability effects that can be reversed. For upgradeable contracts, assume the contract private key is available.

SCOPE (S):

Captures whether a vulnerability in one vulnerable contract impacts resources in other contracts.

METRICS:

SEVERITY COEFFICIENT (C)	COEFFICIENT VALUE	NUMERICAL VALUE
Reversibility (r)	None (R:N) Partial (R:P) Full (R:F)	1 0.5 0.25
Scope (s)	Changed (S:C) Unchanged (S:U)	1.25 1

Severity Coefficient C is obtained by the following product:

$$C = rs$$

The Vulnerability Severity Score S is obtained by:

$$S = min(10, EIC * 10)$$

The score is rounded up to 1 decimal places.

SEVERITY	SCORE VALUE RANGE
Critical	9 - 10
High	7 - 8.9
Medium	4.5 - 6.9
Low	2 - 4.4
Informational	0 - 1.9

5. SCOPE

FILES AND REPOSITORY

(a) Repository: unleash-audit-smart-contract

(b) Assessed Commit ID: aea171b

(c) Items in scope:

- AggregatorV3Interface.sol
- PythAggregator.sol
- PythStructs.sol
- Multicall3.sol
- RedstoneAggregatorV3.sol

Out-of-Scope: Third party dependencies and economic attacks.

FILES AND REPOSITORY

(a) Repository: unleash-audit-smart-contract

(b) Assessed Commit ID: b04421f

(c) Items in scope:

- contracts/lrt-looping-hook/VerioLoopingHook.sol
- contracts/internal-looping-hook/InternalLoopingHook.sol
- contracts/libraries/Pool.sol
- contracts/libraries/UniswapV2Library.sol
- contracts/libraries/BytesLib.sol
- contracts/abstracts/FlashRepaySimpleBase.sol
- contracts/abstracts/Execution.sol
- contracts/abstracts/FlashLoanLoopingHookBase.sol
- contracts/abstracts/TokenPulling.sol
- contracts/dex-looping-hook/DexLoopingHookBase.sol
- contracts/dex-looping-hook/UniversalDexLoopingHook.sol
- contracts/HookConfig.sol
- contracts/flash-repay/FlashRepayFlashLoan.sol
- contracts/flash-repay/UniversalQuoter.sol
- contracts/flash-repay/FlashRepay.sol

Out-of-Scope: Third party dependencies and economic attacks.

FILES AND REPOSITORY

(a) Repository: unleash-audit-smart-contract

(b) Assessed Commit ID: 5cabaf4

(c) Items in scope:

- UiPoolDataProviderV3.sol

Out-of-Scope: Third party dependencies and economic attacks.

FILES AND REPOSITORY

(a) Repository: unleash-audit-smart-contract

(b) Assessed Commit ID: aea171b

(c) Items in scope:

- src/test/MockERC20.sol
- src/test/MockIPStudio.sol
- src/test/NFT.sol
- src/test/MockERC404.sol
- src/test/MockIPCO.sol
- src/libraries/IPad/IPGELib.sol
- src/libraries/IPad/FactoryLib.sol
- src/libraries/IPad/OwnershipTokenLib.sol
- src/libraries/IPad/CreatorLib.sol
- src/libraries/IPad/ConfigLib.sol
- src/libraries/IPad/IPCOLib.sol
- src/libraries/IPad/IPStudioLib.sol
- src/libraries/IPad/PresetLib.sol
- src/libraries/IPStudio/StakeLib.sol
- src/libraries/FullMath.sol
- src/utils/Errors.sol
- src/utils/Events.sol
- src/Presets/IPCO/IPCOHardCapFixedPrice.sol
- src/Presets/IPStudio/Timelock/Timelock.sol
- src/Presets/IPStudio/IPStudioTimelockControlQuorumFraction.sol
- src/Presets/OwnershipToken/UnleashERC404.sol
- src/Presets/OwnershipToken/UnleashERC20.sol

Out-of-Scope: Third party dependencies and economic attacks.

REMEDIATION COMMIT ID:

- 367ff1f
- 388baf5
- https://github.com/unleash-protocol/unleash-audit-smart-contract/blob/6bc964b04ebb90282fdec8753f4779b0f77beb72/contracts/libraries/UniswapV2Library.sol
- 367ff1f

Out-of-Scope: New features/implementations after the remediation commit IDs.

6. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	2	2	2	4

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
MALICIOUS DELEGATEES CAN BLOCK ALL DELEGATORS FUNDS	HIGH	SOLVED - 01/22/2025
INITIAL PROPOSAL PASSES WITH MINIMAL STAKE	HIGH	SOLVED - 02/05/2025
CONFIDENCE INTERVALS FROM PYTH ORACLE ARE IGNORED	MEDIUM	RISK ACCEPTED - 02/10/2025
MISSING ZERO RESERVES CHECK IN UNISWAPV2LIBRARY	MEDIUM	SOLVED - 01/22/2025
UNRESTRICTED PYTH PRICE UPDATES	LOW	NOT APPLICABLE

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
INCORRECT CEI PATTERN IMPLEMENTATION IN STAKELIB'S UNSTAKE()	LOW	SOLVED - 01/22/2025
USE OF UNLICENSED SMART CONTRACTS	INFORMATIONAL	SOLVED - 01/22/2025
USE OF CUSTOM ERRORS INSTEAD OF REVERT STRINGS	INFORMATIONAL	SOLVED - 01/22/2025
INTEGRATION WITH UNDOCUMENTED PROTOCOL	INFORMATIONAL	ACKNOWLEDGED - 02/05/2025
UNUSED IMPORTS	INFORMATIONAL	SOLVED - 01/22/2025

7. FINDINGS & TECH DETAILS

7.1 MALICIOUS DELEGATEES CAN BLOCK ALL DELEGATORS FUNDS

// HIGH

Description

The **IPStudioTimelockControlQuorumFraction** contract is a governance mechanism that empowers users to stake ERC20 tokens to gain voting power, unstake those tokens when desired, and delegate their accrued voting power to another address. This delegation system allows users to either participate directly or offer their voting rights to delegates who can act on their behalf. Internally, the contract uses the **StakeLib** library, which is responsible for managing users' stake balances, recording delegations, and maintaining a history of vote checkpoints over time.

One the user acquired a voting power, they are able to delegate it to another user (delegatee). This delegation is intended to be changeable or removable by the user at any time, allowing dynamic governance participation.

Within the **StakeLib** library, the delegation process is handled by the delegate function. It retrieves the current delegate of the delegator using the delegates function, updates the delegate to the new delegatee and moves the voting power from the old delegate to the new one:

```
95  function delegate(
96      mapping(address => address) storage _delegatee,
97      mapping(address => Checkpoints.Trace208) storage _delegateCheckpoir
98      mapping(address => uint256) storage stakes,
99      address delegator,
100     address delegatee,
101     uint48 clock
102 )
103     external
104 {
105     address oldDelegate = delegates(_delegatee, delegator);
106     _delegatee[delegator] = delegatee;
107
108     emit Events.DelegateChanged(delegator, oldDelegate, delegatee);
109     _moveDelegateVotes(_delegateCheckpoints, oldDelegate, delegatee, _g
110 }
```

The delegates function retrieves the delegatee for a given account. If the delegatee has never been set (i.e. `_delegatee[account] == address(0)`), it defaults to self-delegation by returning the account itself.

This means that the system treats the account as if it has delegated to itself. This fallback is intended as a convenience feature, allowing users to have voting power without explicitly delegating it:

```
function delegates(mapping(address => address) storage _delegatee, addr
    address delegatee = _delegatee[account];
    return delegatee == address(0) ? account : _delegatee[account];
}
```

This mechanism introduces a vulnerability that can be exploited to disrupt the governance system. It arises from the way the contract handles delegations to `address(0)` and how it interacts with the `_moveDelegateVotes` function:

- Inside the `delegate` function of **StakeLib** library, the `oldDelegate` is determined by calling the delegates function. If `_delegatee[delegator]` is already `address(0)`, the fallback causes delegates to return the delegator itself.
- The `delegatee` mapping is updated to the new delegatee (`address(0)` in this case).
- The `_moveDelegateVotes` function is called to transfer voting power from the `oldDelegate` to the new `delegatee`:

```
function _moveDelegateVotes(
    mapping(address => Checkpoints.Trace208) storage _delegateCheckpoir
    address from,
    address to,
    uint256 amount,
    uint48 clock
)
    private
{
    if (from != to && amount > 0) {
        if (from != address(0)) {
            (uint256 oldValue, uint256 newValue) =
                _push(_delegateCheckpoints[from], _subtract, SafeCast.t
            emit Events.DelegateVotesChanged(from, oldValue, newValue);
        }
        if (to != address(0)) {
            (uint256 oldValue, uint256 newValue) =
                _push(_delegateCheckpoints[to], _add, SafeCast.toUint20
            emit Events.DelegateVotesChanged(to, oldValue, newValue);
        }
    }
```

- The function first checks if `from != to` and `amount > 0`. Since `from` is the delegator, `to` is `address(0)`, and assuming delegator is not `address(0)`, the condition holds true.
- The function subtracts the voting power from the from delegatee (which is the delegator themselves due to the fallback).

- Because the delegates function always returns the delegator when `delegatee[delegator] == address(0)`, repeatedly delegating to `address(0)` causes `_moveDelegateVotes` to be called with `from = delegator, to = address(0), amount = userVotingPower` multiple times. Each call subtracts the same amount from the delegator's checkpoint.

As presented in the Proof of Concept, after the first delegation to `address(0)`, the attacker's voting power is reduced to zero. Subsequent delegations attempt to subtract the same amount again, causing an integer underflow.
If other users have delegated their votes to the malicious delegatee, their voting power is effectively tied to the delegatee's checkpoint. With the checkpoint causing underflows, these delegators are prevented from transferring or re-delegating their tokens, leading to a loss of all funds.

Proof of Concept

Malicious delegatee can prevent a user from redelegating or unstaking their tokens, effectively locking their funds:

```
function test_AttackerCannotWithdrawDueToDelegateToZero() external {
    (, address _ownershipToken, address _ipstudio,) = createCollection(4, 1

    //Assign 100 tokens to user and attacker
    deal(_ownershipToken, user, 100 ether);
    deal(_ownershipToken, attacker, 100 ether);

    IERC20 ownershipToken = IERC20(_ownershipToken);
    IPStudioTimelockControlQuorumFraction ipstudio = IPStudioTimelockContro

    //User approves and stakes 100 tokens, then delegates votes to attacker
    vm.startPrank(user);
        ownershipToken.approve(_ipstudio, 100 ether);
        ipstudio.stake(100 ether);
        ipstudio.delegate(attacker);
    vm.stopPrank();

    //Attacker approves and stakes 100 tokens, then delegates their votes t
    vm.startPrank(attacker);
        ownershipToken.approve(_ipstudio, 100 ether);
        ipstudio.stake(100 ether);
        ipstudio.delegate(address(0));

        //Attacker unstake all their tokens
        ipstudio.unstake(100 ether);
    vm.stopPrank();

    //User cannot redelegate their votes nor unstake tokens
    vm.startPrank(user);
        vm.expectRevert();
```

```
            ipstudio.delegate(user);

        vm.expectRevert();
            ipstudio.unstake(100 ether);
    vm.stopPrank();
}
```

Result:

```
Ran 1 test for test/IPStudio.sol/IPStudio.t.sol:IPStudioTest
[PASS] test_AttackerCannotWithdrawDueToDelegateToZero() (gas: 4172611)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 6.63s (2.04s CPU time)

Ran 1 test suite in 6.63s (6.63s CPU time): 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

BVSS

AO:A/AC:L/AX:M/R:N/S:U/C:N/A:M/I:N/D:C/Y:M (8.4)

Recommendation

It is recommended to prevent users from delegating to the zero address by implementing regarded require statement during delegation flow.

Remediation

SOLVED : The **Unleash team** added a check to ensure `delegatee != address(0)`

Remediation Hash

https://github.com/unleash-protocol/unleash-audit-smart-contract/commit/367ff1fb3f029b0239bd4d7c9931ca76467e6568

7.2 INITIAL PROPOSAL PASSES WITH MINIMAL STAKE

// HIGH

Description

Due to the absence of a proper zero total supply check in `IPStudioTimelockControlQuorumFraction.sol`, a malicious user is able to pass any proposal with minimal stake, bypassing the intended quorum requirement. The vulnerability exists in the quorum calculation:

```
function quorum(uint256 timepoint) public view override(GovernorUpgradeable
    return StakeLib.getPastTotalStaked(_totalCheckpoints, timepoint) * quor
}
```

When the contract is initially deployed and no one has staked yet, `_totalCheckpoints` is empty, causing `getPastTotalStaked()` to return `0`. This results in the quorum calculation evaluating to `0`, allowing any proposal to pass regardless of the configured quorum requirement.

This vulnerability allows an attacker to:
1. Pass malicious proposals with minimal stake (just enough to meet the proposal threshold)
2. Execute arbitrary governance actions through the timelock
3. Completely bypass the intended quorum requirements
4. Take control of the governance system before legitimate users can participate

Proof of Concept

Add this test to `IPStudio.t.sol`:

```
//E forge test --match-test "test_proposalPassWithMinimalStake" -vv
    function test_proposalPassWithMinimalStake() external {
        // Create collection with high quorum numerator (40%) to demonstrat
        (address collection, address _ownershipToken, address _ipstudio,) =
            createCollection(40, 1000, 100, 10 ether, 100, "Test");

        // Only the malicious user (accounts[0]) gets tokens and stakes
        deal(_ownershipToken, accounts[0], 10 ether);

        IERC20 ownershipToken = IERC20(_ownershipToken);
        IPStudioTimelockControlQuorumFraction ipstudio = IPStudioTimelockCo
        TimelockControllerUpgradeable timelock = TimelockControllerUpgradea

        // Malicious user stakes just enough to meet proposal threshold
        vm.startPrank(accounts[0]);
        ownershipToken.approve(_ipstudio, 10 ether);
        ipstudio.stake(10 ether);
        vm.roll(vm.getBlockNumber() + 1);
```

```
        // Create malicious proposal
        address[] memory targets = new address[](1);
        targets[0] = REGISTRATION_WORKFLOW;
        uint256[] memory values = new uint256[](1);
        values[0] = 0;
        bytes[] memory data = new bytes[](1);
        data[0] = abi.encodeWithSelector(
            IRegistrationWorkflows.mintAndRegisterIp.selector,
            collection,
            address(timelock),
            WorkflowStructs.IPMetadata({
                ipMetadataURI: "malicious_uri",
                ipMetadataHash: "0x0",
                nftMetadataURI: "malicious_nft_uri",
                nftMetadataHash: "0x0"
            })
        );

        uint256 proposalId = ipstudio.propose(targets, values, data, "Malic

        // Wait for voting delay
        vm.roll(ipstudio.proposalSnapshot(proposalId) + 1);

        // Despite having only minimal stake (10 ether), the vote will pass
        // This shouldn't work with a 40% quorum requirement, but it does d
        ipstudio.castVote(proposalId, uint8(GovernorCountingSimpleUpgradeab

        // Wait for voting period to end
        vm.roll(ipstudio.proposalDeadline(proposalId) + 1);

        // Show that proposal succeeded despite not meeting the intended 40
        assertEq(uint8(ipstudio.state(proposalId)), uint8(IGovernor.Proposa

        // Can queue and execute the malicious proposal
        ipstudio.queue(targets, values, data, keccak256(bytes("Malicious Pr
        assertEq(uint8(ipstudio.state(proposalId)), uint8(IGovernor.Proposa

        vm.warp(ipstudio.proposalEta(proposalId) + 1);
        ipstudio.execute(targets, values, data, keccak256(bytes("Malicious

        // Verify malicious action succeeded
        assertEq(IERC721(collection).ownerOf(1), address(timelock));
    }
```

Result:

```
Encountered a total of 1 failing tests, 0 tests succeeded
➜  unleash-audit-smart-contract-unleash-ipad export STORY_RPC=https://rpc.odyssey.storyrpc.io/

➜  unleash-audit-smart-contract-unleash-ipad forge test --match-test "test_proposalPassWithMinimalStake" -vv
[⠊] Compiling...
No files changed, compilation skipped

Ran 1 test for test/IPStudio.sol/IPStudio.t.sol:IPStudioTest
[PASS] test_proposalPassWithMinimalStake() (gas: 4584818)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 15.00s (8.77s CPU time)

Ran 1 test suite in 15.00s (15.00s CPU time): 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

BVSS

AO:A/AC:L/AX:L/C:N/I:M/A:M/D:N/Y:N/R:N/S:C (7.8)

Recommendation

Implement a minimum total stake requirement before allowing proposals:

```
function propose(
    address[] memory targets,
    uint256[] memory values,
    bytes[] memory calldatas,
    string memory description
) public override returns (uint256) {
    require(
        StakeLib.getPastTotalStaked(_totalCheckpoints, block.number - 1) >=
        "Total stake too low"
    );
    return super.propose(targets, values, calldatas, description);
}
```

Remediation

SOLVED : The **Unleash team** added a proposing time to ensure a minimum total stake exist.

Remediation Hash

https://github.com/unleash-protocol/unleash-audit-smart-contract/commit/388baf5941b5def842a7021d3fd241064c034e24

References

unleash-protocol/unleash-audit-smart-contract/src/Presets/IPStudio/IPStudioTimelockControlQuorumFraction.sol#L181

7.3 CONFIDENCE INTERVALS FROM PYTH ORACLE ARE IGNORED

// MEDIUM

Description

The protocol relies on prices provided by the Pyth network but does not incorporate the accompanying confidence intervals. These intervals indicate how certain or uncertain the price might be. According to Pyth's best practices, checking confidence intervals helps guard against using prices that may be unreliable, especially during times of high market volatility or when the price feed encounters data issues.

```
95    function latestRoundData()
96      external
97      view
98      returns (
99        uint80 roundId,
100       int256 answer,
101       uint256 startedAt,
102       uint256 updatedAt,
103       uint80 answeredInRound
104     )
105   {
106     PythStructs.Price memory price = pyth.getPriceUnsafe(priceId);
107     roundId = uint80(price.publishTime);
108     return (roundId, int256(price.price), price.publishTime, price.publ
109   }
```

By ignoring this information, the protocol risks relying on a price estimate with a very wide confidence interval. It could use prices that are less accurate than expected, increasing the risk of incorrect actions such as mispriced trades, incorrect liquidation triggers, or erroneous value calculations. Attackers might manipulate or benefit from a large confidence interval if the protocol continues to accept prices as valid without additional checks.

BVSS

AO:A/AC:L/AX:M/C:N/I:N/A:N/D:H/Y:N/R:N/S:U (5.0)

Recommendation

It is recommended to incorporate checks on the confidence interval to ensure that the price is reliable before using it. For example, ensure that the ratio σ/p (i.e. confidence interval divided by the price) remains below an acceptable threshold.

Remediation

RISK ACCEPTED : The **Unleash team** decided to accept this finding as using Pyth price in low-confidence scenarios is necessary to prevent liquidation from stalling, which could otherwise lead to bad debt.

References

-

7.4 MISSING ZERO RESERVES CHECK IN UNISWAPV2LIBRARY

// MEDIUM

Description

The `UniswapV2Library` contract lacks reserve validation checks in both `getAmountIn()` and `getAmountOut()` functions, enabling trades against pools with zero reserves. This deviates from the original Uniswap V2 implementation, which enforces non-zero reserves.

```
8  function getAmountIn(IUniswapV2Pair pair, address tokenIn, uint256 amou
9      require(amountOut > 0, "UniswapV2Library: INSUFFICIENT_OUTPUT_AMOUN
10
11     (uint256 reserveIn, uint256 reserveOut,) = pair.getReserves();
12     //Missing check: require(reserveIn > 0 && reserveOut > 0)
13     /// ... ///
14 }
```

```
27 function getAmountOut(IUniswapV2Pair pair, address tokenIn, uint256 amo
28     require(amountIn > 0, "UniswapV2Library: INSUFFICIENT_INPUT_AMOUNT'
29
30     (uint256 reserveIn, uint256 reserveOut,) = pair.getReserves();
31     //Missing check: require(reserveIn > 0 && reserveOut > 0)
32     /// ... ///
33 }
```

Official UniV2 Library :

```
function getAmountOut(uint amountIn, uint reserveIn, uint reserveOut) inter
    require(amountIn > 0, 'UniswapV2Library: INSUFFICIENT_INPUT_AMOUNT');
    require(reserveIn > 0 && reserveOut > 0, 'UniswapV2Library: INSUFFICIEN
}
```

Impacts can be big :

1. Price manipulation by executing trades against pools with zero reserves
2. Division by zero errors in `getAmountIn()` when `reserveOut` is 0
3. Incorrect pricing model in `getAmountOut()` when `reserveIn` is 0

BVSS

AO:A/AC:L/AX:L/C:N/I:L/A:L/D:L/Y:N/R:N/S:C (4.7)

Recommendation

It is recommended to add reserve validation checks in both uniswapV2 functions.

Remediation

SOLVED : The **Unleash team** added a check on the reserve amount to ensure there is enough liquidity.

Remediation Hash

https://github.com/unleash-protocol/unleash-audit-smart-contract/blob/6bc964b04ebb90282fdec8753f4779b0f77beb72/contracts/libraries/UniswapV2Library.sol

References

-

7.5 UNRESTRICTED PYTH PRICE UPDATES

// LOW

Description

The **AaveOracle** contract is implemented as oracle interface that expects to read a single valid price per asset from its aggregator (for instance, `PythAggregatorV3`). In this case, the aggregator in turn fetches the price from Pyth each time someone calls it. As Pyth uses a pull model:

- It receives off-chain signed updates for asset prices.
- Anyone can "pull" these updates on-chain by calling `updatePriceFeeds` in the same transaction they are using to do a the intended operation (e.g. deposit or borrow in **Aave** case).

Because the price can be updated within the same transaction in which a user interacts with Aave, this opens the possibility of an instant-arbitrage scenario. A malicious user can:

- Provide old (low) signed data to set the on-chain price to a beneficially low value.
- Provide new (high) signed data—still in the same transaction—to set the on-chain price to a beneficially high value.
- Interact again with Aave, now using the updated and changed price.

Even if it cannot lead to an impactful scenario right now, it leaves an open door for a future exploit.

BVSS

AO:A/AC:L/AX:L/C:N/I:L/A:N/D:N/Y:N/R:N/S:C (3.1)

Recommendation

It is recommended to record a token's price in **AaveOracle** contract, the first time it is accessed in each block and use that same price for all subsequent operations within the same block. This ensures the token's price remains consistent throughout the entire block.

Remediation

NOT APPLICABLE : After discussion with the **Unleash team,** we reached the conclusion that this finding is not applicable to the current Unleash protocol.

References

-

7.6 INCORRECT CEI PATTERN IMPLEMENTATION IN STAKELIB'S UNSTAKE()

// LOW

Description

The `unstake()` function in the StakeLib library violates the Checks-Effects-Interactions (CEI) pattern by performing an external call after a state modification. The external token transfer occurs after updating the stakes mapping but before updating the voting units state, creating a potential reentrancy vulnerability.

```
function unstake(
    /// ... ///
)
    external
{
    require(stakes[user] >= amount, Errors.InsufficientStake());
    stakes[user] -= amount;
    IERC20(token).safeTransfer(user, amount);

    _transferVotingUnits(_delegatee, _delegateCheckpoints, _totalCheckpoint
    emit Events.Unstaked(user, amount);
}
```

BVSS

AO:A/AC:L/AX:L/C:N/I:L/A:N/D:N/Y:N/R:N/S:U (2.5)

Recommendation

It is recommended to modify the `unstake()` function to follow the CEI pattern by moving the token transfer after all state modifications.

Remediation

SOLVED : The **Unleash team** modified the function, which now respects the CEI pattern.

Remediation Hash

https://github.com/unleash-protocol/unleash-audit-smart-contract/commit/367ff1fb3f029b0239bd4d7c9931ca76467e6568#diff-aa509ee30eda430d39dc7c0235b6847e0100e265a01d6fd831bbec0e2befc5fe

References

unleash-protocol/unleash-audit-smart-contract/src/libraries/IPStudio/StakeLib.sol#L37

7.7 USE OF UNLICENSED SMART CONTRACTS

// INFORMATIONAL

Description

All the smart contracts in the **IPAD** repository are marked as unlicensed, as indicated by the SPDX license identifier at the top of the files:

```
// SPDX-License-Identifier: UNLICENSED
```

Using unlicensed contract can lead to legal uncertainties and potential conflicts regarding the usage, modification and distribution rights of the code. This may deter other developers from using or contributing to the project.

Score

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:U (0.0)

Recommendation

It is recommended to choose and apply an appropriate open-source license to the smart contracts. Some popular options for blockchain and smart contract projects include:

1. MIT License: A permissive license that allows for reuse with minimal restrictions.
2. GNU General Public License (GPL): A copyleft license that ensures derivative works are also open-source.
3. Apache License 2.0: A permissive license that provides an express grant of patent rights from contributors to users.

Remediation

SOLVED : The **Unleash team** added a license on all the contracts.

Remediation Hash

https://github.com/unleash-protocol/unleash-audit-smart-contract/commit/367ff1fb3f029b0239bd4d7c9931ca76467e6568#diff-aa509ee30eda430d39dc7c0235b6847e0100e265a01d6fd831bbec0e2befc5fe

7.8 USE OF CUSTOM ERRORS INSTEAD OF REVERT STRINGS

// INFORMATIONAL

Description

Throughout the entire codebases, `require` statements are used. In Solidity development, replacing hard-coded revert message strings with the `Error()` syntax is an optimization strategy that can significantly reduce gas costs. Hardcoded strings, stored on the blockchain, increase the size and cost of deploying and executing contracts. The `Error()` syntax allows for the definition of reusable, parameterized custom errors, leading to a more efficient use of storage and reduced gas consumption.

This approach not only optimizes gas usage during deployment and interaction with the contract but also enhances code maintainability and readability by providing clearer, context-specific error information.

Score

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:U (0.0)

Recommendation

It is recommended to replace all revert strings with custom errors.

Remediation

SOLVED : The **Unleash team** added custom errors on the recommended contracts.

Remediation Hash

https://github.com/unleash-protocol/unleash-audit-smart-contract/commit/367ff1fb3f029b0239bd4d7c9931ca76467e6568#diff-aa509ee30eda430d39dc7c0235b6847e0100e265a01d6fd831bbec0e2befc5fe

7.9 INTEGRATION WITH UNDOCUMENTED PROTOCOL

// INFORMATIONAL

Description

The `VerioLoopingHook` contract integrates with the Verio Protocol, which has no public documentation or accessible source code neither official audit. This creates unverifiable security assumptions in critical parts of the contract. Key areas of concern are found in the following code sections:

```
function loop(
    uint256 amount,
    uint256 borrowAmount,
    uint256 interestRateMode,
    uint256 minVIPAmount
) public payable nonReentrant {
```

The loop function initiates interactions with Verio Protocol's staking mechanism combined with flash loans, creating an integration point with unknown implementation details.

```
function _depositToPool(
    address user,
    uint256 amount,
    uint256 minVIPAmount
) internal returns (bool) {
    // Deposit IP to Verio
    IVerioStakePool verioPool = VERIO_POOL;
    uint256 vIPAmount = verioPool.calculateVIPMint(amount);
    require(vIPAmount >= minVIPAmount, "Insufficient vIP amount");
    verioPool.stake{value: amount}();
```

This section demonstrates several trust assumptions:

1. `calculateVIPMint()` - Calls an opaque pricing function with unknown implementation
2. `stake()` - Sends value to an undocumented staking mechanism
3. `getVIP()` - Trusts an unknown token implementation

The contract relies on these closed-source functions for:

- Token minting calculations
- Value staking operations
- Token behavior assumptions
- Price calculations

Score

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:C (0.0)

Recommendation

It is recommended to not integrate with non public and non audited protocol.

Remediation

ACKNOWLEDGED : The **Unleash team** provided the relevant code snippets for the functions in use, and no vulnerabilities were identified in the hook contract unless they originate from the Verio protocol itself. This assessment does not constitute a full audit of the Verio protocol.

References

unleash-protocol/unleash-audit-smart-contract/contracts/lrt-looping-hook/VerioLoopingHook.sol#L13

7.10 UNUSED IMPORTS

// INFORMATIONAL

Description

Throughout the code in scope, there are several instances where the imports are declared but never used:

- `core` repository:
  - In `PriceFeedManagerEvents.sol`:
    - `import {IRedstoneAdapter} from '@redstone-finance/on-chain-relayer/contracts/core/IRedstoneAdapter.sol';`

- `IPAD` repository:
  - In `StakeLib.sol` :
    - `import { Creator } from "../../interfaces/IIPad.sol";`
    - `import { Votes } from "@openzeppelin/contracts/governance/utils/Votes.sol";`
    - `import { console2 } from "forge-std/src/console2.sol";`
  - In `IPStudioTimelockControlQuorumFraction.sol` :
    - `import { IIPad } from "../../interfaces/IIPad.sol";`
    - `import { Initializable } from "@openzeppelin/contracts-upgradeable/proxy/utils/Initializable.sol";`
  - In `Ipad.sol`:
    - `import { console2 } from "forge-std/src/console2.sol";`

- `hooks` repository:
  - In `UniversalDexLoopingHook.sol`:
    - `import {IWNative} from "../interfaces/IWNative.sol";`
  - In `FlashRepay.sol`:
    - `import {DataTypes, TokenApproveSig, FlashRepaySimpleBase} from "../abstracts/FlashRepaySimpleBase.sol";`
  - In `FlashRepayLoan.sol`:
    - `import {DataTypes, TokenApproveSig, FlashRepaySimpleBase} from "../abstracts/FlashRepaySimpleBase.sol";`

Score

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:C (0.0)

Recommendation

It is recommended to remove the unused imports.

Remediation

SOLVED : The **Unleash team** removed the unused imports.

Remediation Hash

https://github.com/unleash-protocol/unleash-audit-smart-contract/commit/367ff1fb3f029b0239bd4d7c9931ca76467e6568#diff-aa509ee30eda430d39dc7c0235b6847e0100e265a01d6fd831bbec0e2befc5fe

8. AUTOMATED TESTING

Halborn used automated testing techniques to enhance the coverage of certain areas of the smart contracts in scope. Among the tools used was Slither, a Solidity static analysis framework.
After Halborn verified the smart contracts in the repository and was able to compile them correctly into their abis and binary format, Slither was run against the contracts. This tool can statically verify mathematical relationships between Solidity variables to detect invalid or inconsistent usage of the contracts' APIs across the entire code-base.

```
INFO:Detectors:
IPStudioTimelockControlQuorumFraction.initialize(address,address,IIPInitialize.IPInitializeData,bytes) (src/Presets/IPStudio/IPStudioTimelockControlQuorumFraction.sol#73-106) uses arbitrary from in transferFrom: IE
RC721(tokenContract).transferFrom(address(this),address(timelock),tokenId) (src/Presets/IPStudio/IPStudioTimelockControlQuorumFraction.sol#98)
StakeLib.stake(mapping(address => uint256),mapping(address => address),mapping(address => Checkpoints.Trace208),Checkpoints.Trace208,address,address,uint256,uint48) (src/libraries/IPStudio/StakeLib.sol#18-35) uses
arbitrary from in transferFrom: IERC20(token).safeTransferFrom(user,address(this),amount) (src/libraries/IPStudio/StakeLib.sol#31)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#arbitrary-from-in-transferfrom
INFO:Detectors:
GovernorUpgradeable._executeOperations(uint256,address[],uint256[],bytes[],bytes32) (node_modules/@openzeppelin/contracts-upgradeable/governance/GovernorUpgradeable.sol#470-481) sends eth to arbitrary user
        Dangerous calls:
        - (success,returndata) = targets[i].call{value: values[i]}(calldatas[i]) (node_modules/@openzeppelin/contracts-upgradeable/governance/GovernorUpgradeable.sol#478)
GovernorUpgradeable.relay(address,uint256,bytes) (node_modules/@openzeppelin/contracts-upgradeable/governance/GovernorUpgradeable.sol#692-695) sends eth to arbitrary user
        Dangerous calls:
        - (success,returndata) = target.call{value: value}(data) (node_modules/@openzeppelin/contracts-upgradeable/governance/GovernorUpgradeable.sol#693)
TimelockControllerUpgradeable._execute(address,uint256,bytes) (node_modules/@openzeppelin/contracts-upgradeable/governance/TimelockControllerUpgradeable.sol#437-440) sends eth to arbitrary user
        Dangerous calls:
        - (success,returndata) = target.call{value: value}(data) (node_modules/@openzeppelin/contracts-upgradeable/governance/TimelockControllerUpgradeable.sol#438)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#functions-that-send-ether-to-arbitrary-destinations
INFO:Detectors:
Math.mulDiv(uint256,uint256,uint256) (node_modules/@openzeppelin/contracts/utils/math/Math.sol#144-223) has bitwise-xor operator ^ instead of the exponentiation operator **:
         - inverse = (3 * denominator) ^ 2 (node_modules/@openzeppelin/contracts/utils/math/Math.sol#205)
FullMath.mulDiv(uint256,uint256,uint256) (src/libraries/FullMath.sol#17-105) has bitwise-xor operator ^ instead of the exponentiation operator **:
         - inv = (3 * denominator) ^ 2 (src/libraries/FullMath.sol#85)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-exponentiation
INFO:Detectors:
IPad.collections (src/IPad.sol#52) is never initialized. It is used in:
        - IPad.collectionByAddress(address) (src/IPad.sol#101-103)
IPad.collectionIndexs (src/IPad.sol#53) is never initialized. It is used in:
        - IPad.collectionByAddress(address) (src/IPad.sol#101-103)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#uninitialized-state-variables
```

```
INFO:Detectors:
FlashRepaySimpleBase.withdrawAToken(address,uint256,address) (src/abstracts/FlashRepaySimpleBase.sol#46-59) uses arbitrary from in transferFrom: IERC20(reserveData.aTokenAddress).safeTransferFrom(from,address(this)
,repayAmount) (src/abstracts/FlashRepaySimpleBase.sol#49)
FlashRepaySimpleBase.withdrawAToken(address,uint256,address) (src/abstracts/FlashRepaySimpleBase.sol#46-59) uses arbitrary from in transferFrom: IERC20(reserveData.aTokenAddress).safeTransferFrom(from,address(this)
,ONE) (src/abstracts/FlashRepaySimpleBase.sol#53)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#arbitrary-from-in-transferfrom
INFO:Detectors:
VerioLoopingHook._depositToPool(address,uint256,uint256) (src/lrt-looping-hook/VerioLoopingHook.sol#115-132) sends eth to arbitrary user
        Dangerous calls:
        - verioPool.stake{value: amount}() (src/lrt-looping-hook/VerioLoopingHook.sol#124)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#functions-that-send-ether-to-arbitrary-destinations
INFO:Detectors:
FlashRepay.swapV2Callback(address,uint256,uint256,bytes) (src/flash-repay/FlashRepay.sol#214-232) ignores return value by IERC20(IUniswapV2Pair(pool).token0()).transfer(data.recipientCached,amount0) (src/flash-repa
y/FlashRepay.sol#224)
FlashRepay.swapV2Callback(address,uint256,uint256,bytes) (src/flash-repay/FlashRepay.sol#214-232) ignores return value by IERC20(IUniswapV2Pair(pool).token1()).transfer(data.recipientCached,amount1) (src/flash-repa
y/FlashRepay.sol#226)
FlashRepayFlashLoan.swapV2Callback(address,uint256,uint256,bytes) (src/flash-repay/FlashRepayFlashLoan.sol#225-242) ignores return value by IERC20(IUniswapV2Pair(pool).token0()).transfer(data.recipientCached,amount
0) (src/flash-repay/FlashRepayFlashLoan.sol#235)
FlashRepayFlashLoan.swapV2Callback(address,uint256,uint256,bytes) (src/flash-repay/FlashRepayFlashLoan.sol#225-242) ignores return value by IERC20(IUniswapV2Pair(pool).token1()).transfer(data.recipientCached,amount
1) (src/flash-repay/FlashRepayFlashLoan.sol#237)
VerioLoopingHook.loop(uint256,uint256,uint256,uint256) (src/lrt-looping-hook/VerioLoopingHook.sol#31-62) ignores return value by WNATIVE.transferFrom(msg.sender,address(this),amount) (src/lrt-looping-hook/VerioLoop
ingHook.sol#47)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#unchecked-transfer
```

All issues identified by Slither were proved to be false positives or have been added to the issue list in this report.

Halborn strongly recommends conducting a follow-up assessment of the project either within six months or immediately following any material changes to the codebase, whichever comes first. This approach is crucial for maintaining the project's integrity and addressing potential vulnerabilities introduced by code modifications.