Persistence - pStake & stkETH

Smart Contract Security Assessment

Prepared by: **Halborn**

Date of Engagement: **May 1st, 2023 - July 14th, 2023**

Visit: **Halborn.com**

DOCUMENT REVISION HISTORY

VERSION	MODIFICATION	DATE	AUTHOR
0.1	Document Creation	05/15/2023	Ataberk Yavuzer
0.2	Document Updates	05/15/2023	Ataberk Yavuzer
0.3	Draft Review	05/16/2023	Gokberk Gulgun
0.4	Draft Review	05/16/2023	Gabi Urrutia
1.0	Second Assessment Updates	06/22/2023	Ataberk Yavuzer
1.1	Second Assessment Updates	07/12/2023	Grzegorz Trawinski
1.2	Second Assessment Updates	07/14/2023	Grzegorz Trawinski
1.3	Second Assessment Updates Review	07/17/2023	Piotr Cielas
2.0	Remediation Plan	07/20/2023	Grzegorz Trawinski
2.1	Remediation Plan Review	07/21/2023	Gabi Urrutia

CONTACTS

CONTACT	COMPANY	EMAIL
Rob Behnke	Halborn	Rob.Behnke@halborn.com
Steven Walbroehl	Halborn	Steven.Walbroehl@halborn.com
Gabi Urrutia	Halborn	Gabi.Urrutia@halborn.com
Gokberk Gulgun	Halborn	Gokberk.Gulgun@halborn.com
Ataberk Yavuzer	Halborn	Ataberk.Yavuzer@halborn.com
Grzegorz Trawinski	Halborn	Grzegorz.Trawinski@halborn.com
Piotr Cielas	Halborn	Piotr.Cielas@halborn.com

EXECUTIVE OVERVIEW

1.1 INTRODUCTION

Persistence is a Tendermint-based specialized Layer-1 powering an ecosystem of DeFi applications focused on unlocking the liquidity of staked assets.

Persistence engaged Halborn to conduct a security assessment on their smart contracts beginning on May 1st, 2023 and ending on July 14th, 2023. The security assessment was scoped to the smart contracts provided to the Halborn team.

Several code updates with remediation and functionality fixes were delivered after 22th of June with final commit set to a39243693fdf0d08dafc0dbfe5e01886c6299d3b. Halborn performed a security review of the new updates between 22th of June and 14th of July.

1.2 ASSESSMENT SUMMARY

The team at Halborn was provided two weeks for the engagement and assigned a full-time security engineer to review the security of the smart contracts. The security engineer is a blockchain and smart-contract security expert with advanced penetration testing and smart-contract hacking skills, and deep understanding of multiple blockchain protocols.

For the updates review, Halborn was provided 3 weeks.

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 security risks that were partially addressed by the Persistence team.

1.3 SCOPE

1. Persistence - pStake and stkETH Smart Contract Repository
   (a) Repository: pstake-stkETH
   (b) Commit ID: 00a239d4bd72db83c293834e2c90c21060e7469d
2. In-Scope:
   (a) L1-contracts/contracts/Core.sol
   (b) L1-contracts/contracts/CoreRef.sol
   (c) L1-contracts/contracts/IssuerUpgradable.sol
   (d) L1-contracts/contracts/KeysManager.sol
   (e) L1-contracts/contracts/L1MessageContract.sol
   (f) L1-contracts/contracts/Oracle.sol
   (g) L1-contracts/contracts/Permissions.sol
   (h) L1-contracts/contracts/StakingPool.sol
   (i) L1-contracts/contracts/WithdrawalCredential.sol
   (j) L1-contracts/contracts/token/StkEth.sol
   (k) L1-contracts/contracts/messenger/L1MessengerBase.sol
   (l) L1-contracts/contracts/messenger/OptimismMessenger.sol
   (m) L1-contracts/contracts/interfaces/*
   (n) L2-contracts/contracts/Issuer.sol
   (o) L2-contracts/contracts/L2MessageContract.sol
   (p) L2-contracts/contracts/StkEth.sol
   (q) L2-contracts/contracts/interfaces/*
3. Out-of-Scope:
   (a) L1-contracts/contracts/PriceOracle.sol
   (b) L1-contracts/contracts/mocks/*
   (c) L1-contracts/contracts/testContractsFrontend/*
   (d) L2-contracts/contracts/testContractsFrontend/*

After the findings of the first assessment were resolved, second commit containing new features was sent to Halborn by the Persistence team for a follow-up review.

1. Repository: pstake-stkETH
2. Second Commit ID: a39243693fdf0d08dafc0dbfe5e01886c6299d3b
3. In-Scope:
    (a) L1-contracts/contracts/Core.sol
    (b) L1-contracts/contracts/CoreRef.sol
    (c) L1-contracts/contracts/IssuerUpgradable.sol
    (d) L1-contracts/contracts/KeysManager.sol
    (e) L1-contracts/contracts/Oracle.sol
    (f) L1-contracts/contracts/Permissions.sol
    (g) L1-contracts/contracts/StakingPool.sol
    (h) L1-contracts/contracts/WithdrawalCredential.sol
    (i) L1-contracts/contracts/token/StkEth.sol
    (j) L1-contracts/contracts/messenger/L1MessengerBase.sol
    (k) L1-contracts/contracts/messenger/OptimismMessenger.sol
    (l) L1-contracts/contracts/messenger/ArbitrumMessenger.sol
    (m) L1-contracts/contracts/library/BeaconData.sol
    (n) L1-contracts/contracts/interfaces/*
    (o) L2-contracts/contracts/Issuer.sol
    (p) L2-contracts/contracts/StkEth.sol
    (q) L2-contracts/contracts/arbitrum/L2MessageContractArbitrum.sol
    (r) L2-contracts/contracts/optimism/L2MessageContractOptimism.sol
    (s) L2-contracts/contracts/interfaces/*

On July 20th, 2023, the team at Halborn received the final code base with applied remediation, commit ID: ff40bb442aba920eb90542b9292cb66a8f6e3012.

1.4 TEST APPROACH & METHODOLOGY

Halborn performed a combination of manual and automated security testing to balance efficiency, timeliness, practicality, and accuracy regarding the scope of the smart contract assessment. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance coverage of smart contracts and can quickly identify items that don't follow security best practices. The following phases and associated tools were used throughout the term of the review:

- Research into architecture and purpose.
- Smart Contract manual code review and walkthrough.
- Graphing out functionality and contract logic/connectivity/functions(solgraph)
- Manual Assessment of use and safety for the critical Solidity variables and functions in scope to identify any arithmetic related vulnerability classes.
- Dynamic Analysis & Testing (foundry).
- Local Deployment (anvil).
- Static Analysis (slither, MythX).

2. 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.

2.1 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
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:

$$E = \prod m_e$$

2.2 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)	0
	Low (I:L)	0.25
	Medium (I:M)	0.5
	High (I:H)	0.75
	Critical (I:C)	1
Integrity (I)	None (I:N)	0
	Low (I:L)	0.25
	Medium (I:M)	0.5
	High (I:H)	0.75
	Critical (I:C)	1
Availability (A)	None (A:N)	0
	Low (A:L)	0.25
	Medium (A:M)	0.5
	High (A:H)	0.75
	Critical	1
Deposit (D)	None (D:N)	0
	Low (D:L)	0.25
	Medium (D:M)	0.5
	High (D:H)	0.75
	Critical (D:C)	1
Yield (Y)	None (Y:N)	0
	Low (Y:L)	0.25
	Medium: (Y:M)	0.5
	High: (Y:H)	0.75
	Critical (Y:H)	1

Impact I is calculated using the following formula:

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

2.3 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.

Coefficient (C)	Coefficient Value	Numerical Value
Reversibility (r)	None (R:N)	1
	Partial (R:P)	0.5
	Full (R:F)	0.25
Scope (s)	Changed (S:C)	1.25
	Unchanged (S:U)	1

Severity Coefficient C is obtained by the following product:

$$C = rs$$

EXECUTIVE OVERVIEW

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

3. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	1	4	5	7

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
(HAL-01) GETDEPOSITL2 MAY REVERT DUE TO INCORRECT CHECK OF EXCHANGE RATE ERROR CONDITION	High (7.0)	SOLVED - 07/14/2023
(HAL-02) STAKERS CAN LOSE THEIR ASSETS DUE TO PRICE SHARE MANIPULATION	Medium (6.8)	SOLVED - 06/16/2023
(HAL-03) MALICIOUS ORACLE MEMBER CAN SLASH ALL STAKES FROM THE PROTOCOL	Medium (6.3)	SOLVED - 06/16/2023
(HAL-04) LOSS OF FUNDS DUE TO MISSED LOGICAL IMPLEMENTATIONS	Medium (5.6)	RISK ACCEPTED
(HAL-05) EXCESSIVELY CENTRALIZED FUNCTIONALITY	Medium (5.0)	FUTURE RELEASE
(HAL-06) IMPROPER IMPLEMENTATION OF TRANSFERETHMAINNET FUNCTION	Low (3.9)	SOLVED - 06/22/2023
(HAL-07) USE OF OPTIMISM TESTNET CHAIN ID	Low (3.8)	FUTURE RELEASE
(HAL-08) GETDEPOSITOPTIMISM FUNCTION ALWAYS GETS REVERTED	Low (3.1)	SOLVED - 06/22/2023
(HAL-09) USE OF TRANSFER/TRANSFERFROM METHOD INSTEAD OF SAFETRANSFER/SAFETRANSFERFROM	Low (2.5)	SOLVED - 06/22/2023
(HAL-10) IMPLEMENTATIONS CAN BE INITIALIZED	Low (2.5)	SOLVED - 07/20/2023
(HAL-11) FLOATING PRAGMA	Informational (0.0)	SOLVED - 06/22/2023
(HAL-12) A MESSENGER CAN BE ADDED MORE THAN ONCE	Informational (0.0)	FUTURE RELEASE
(HAL-13) FOR LOOP OPTIMIZATIONS	Informational (0.0)	SOLVED - 06/22/2023
(HAL-14) REDUNDANT LOGICS	Informational (0.0)	SOLVED - 06/22/2023

(HAL-15) UNUSED IMPORTS, VARIABLES AND FUNCTIONS	Informational (0.0)	SOLVED - 06/22/2023
(HAL-16) OPEN TODOS	Informational (0.0)	SOLVED - 06/22/2023
(HAL-17) STRICTLY PACKED VARIABLES CONSUMES LESS GAS	Informational (0.0)	SOLVED - 07/12/2023

FINDINGS & TECH DETAILS

4.1 (HAL-01) GETDEPOSITL2 MAY REVERT DUE TO INCORRECT CHECK OF EXCHANGE RATE ERROR CONDITION - HIGH (7.0)

Description:

The getDepositL2() function within the IssuerUpgradable contract receives ether from L2 sent by socket receiver. However, the exchange rate error condition has an incorrect check implemented, as it reverts whenever the exchange rate is between minimum and maximum bounds value. Thus, it rejects valid transfers.

Code Location:

Listing 1: IssuerUpgradable.sol (Lines 263-264)

```
256     function getDepositL2(
257         uint256 _stkEthMinted,
258         uint256 _sourceChainId
259     ) external payable onlySocketReceiver {
260         // accept 1% error in exchange rate due to delay in bridging
261         uint256 exchangeRate = core.stkEth().pricePerShare();
262         if (
263             exchangeRate - exchangeRate / 100 <= (msg.value / _stkEthMinted) &&
264             (msg.value / _stkEthMinted) <= exchangeRate + exchangeRate / 100
265         ) revert InvalidExchangeRateReceived();
266         if (msg.value > 0) {
267             ethStaked += msg.value;
268             stkEthMinted += _stkEthMinted;
269             emit EthReceived(msg.value, _stkEthMinted, _sourceChainId);
270         }
271     }
```

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:C/D:M/Y:N/R:P/S:C (7.0)

Recommendation:

It is recommended to fix the exchange rate error condition check within the getDepositL2() function.

Remediation Plan:

SOLVED: The Persistence team solved this issue by implementing correct exchange rate error condition check.

Commit ID: f73adcd22820668a33419df840ab04392477f8ac

4.2 (HAL-02) STAKERS CAN LOSE THEIR ASSETS DUE TO PRICE SHARE MANIPULATION - MEDIUM (6.8)

Description:

During the first deployment of the protocol, the price share is open to manipulation. When the Persistence protocol calculates the price share, it uses the following formula:

Listing 2

```
pricePerShare = ((withdrawals.getTotalRewards() + issuer.ethStaked
    () - withdrawals.getTotalSlashedAmount() - valEthShare -
    protocolEthShare) * 1e18) / issuer.stkEthMinted();
```

That means, the price per share variable is getting calculated with the following parameters:

- Total rewards in the protocol (affects price per share in a positive way).
- Total of staked ETHs (affects price per share in a positive way).
- Total of slashed stakes (affects price per share in a negative way).
- Total of minted ethStk tokens (affects price per share in both positive and negative ways).

Basically, if we increase the total rewards and keep the staked ETH at minimum, the pricePerShare variable will be abnormally high. Therefore, any attacker can **front-run** the first stake and increase the pricePerShare variable.

As a result of this attack, stakers can lose their assets when they are staking.

Proof of Concept:

Listing 3: Front-running shares PoC

```
function test_frontrunningSharesPoC() public {
    // attacker frontruns calls below
    // vm.prank(user3);
    // issuerContract.stake{value: 32 ether}();

    vm.startPrank(attacker);
    (bool success,) = payable(withdrawalAddress).call{value: 1 ether}("");
    require(success);
    issuerContract.stake{value: 1 wei}();
    oracleContract.changeCValue();
    issuerContract.stake{value: 1 ether}();
    vm.stopPrank();

    vm.prank(user3);
    issuerContract.stake{value: 0.9 ether}(); // user3 will lose 0.9 ether for 0 stakes.

    vm.prank(user2);
    issuerContract.stake{value: 0.5 ether}();
    vm.prank(user2);
    issuerContract.stake{value: 0.5 ether}(); // user2 will lose 1 ether for 0 stakes.
}
```

Code Location:

Listing 4: Oracle.sol (Line 134)

```
function changeCValue() public override whenNotPaused {
        int256 calculatedRewards = int256(withdrawals.getNewRewards()) -
            int256(withdrawals.getNewSlashedAmount());
        if (calculatedRewards > 0) {
            uint256 valEthShare = (valCommission * uint256(calculatedRewards)) / BASIS_POINT;
            uint256 protocolEthShare = (pStakeCommission * uint256(calculatedRewards)) /
```

```
                    BASIS_POINT;
                IIssuer issuer = IIssuer(core().issuer());
                pricePerShare =
                    ((withdrawals.getTotalRewards() +
                        issuer.ethStaked() -
                        withdrawals.getTotalSlashedAmount() -
                        valEthShare -
                        protocolEthShare) * 1e18) /
                    issuer.stkEthMinted();
                withdrawals.setNewRewardsToZero();
                withdrawals.distributeRewards(protocolEthShare, valEthShare, pricePerShare);
                // get messengers list to update cValue on L2s
                uint256 numberMessengers = issuer.getNumberMessengers();
                for (uint256 i = 0; i < numberMessengers; i++) {
                    (bool messengerStatus, address messenger) = issuer.getMessenger(i);
                    if (messengerStatus && messenger != address(0)) {
                        IL1Messenger(messenger).changeCValueL2(pricePerShare);
                    }
                }
```

BVSS:

AO:A/AC:M/AX:L/C:N/I:N/A:L/D:H/Y:N/R:N/S:C (6.8)

Recommendation:

In order to prevent such scenarios, it is recommended to mint a certain amount of shares to address(0) when the protocol is first deployed. Also, there should be a lower limit to prevent staking very small amounts.

Remediation Plan:

SOLVED: The Persistence team solved this issue by implementing a lower bound(minimumStakeAmount) for mint and burn operations.

Commit ID: 424e44eb7d15e531487c60099962a6206ef27088

4.3 (HAL-03) MALICIOUS ORACLE MEMBER CAN SLASH ALL STAKES FROM THE PROTOCOL - MEDIUM (6.3)

Description:

During Role-based Access Control tests, a security flaw was detected if the Quorum on the protocol is one which is the default setting for the protocol. Basically, all oracle members have the right to slash stakes for node operators. In the worst-case scenario, there should be three oracle members on the protocol and the quorum should be at least two. In other case, the quorum logic can be bypassed, and malicious oracle members can slash huge amount of staked assets to decrease the pricePerShare variable.

This may result in the malicious oracle member getting a very high number of shares by reducing the price per share to 1 thanks to a malicious slash() call. As a result of this situation, the shares held by other users will lose their value.

Proof of Concept:

Listing 5: Malicious Oracle PoC

```
function testFail_maliciousOracleSlashingPoC() public {
        bytes[] memory _publicKeys = new bytes[](2);
        _publicKeys[0] = pubKeyValidator1;
        _publicKeys[1] = pubKeyValidator2;

        vm.prank(oracleMember2);
        oracleContract.activateValidator(_publicKeys);

        vm.prank(user1);
        issuerContract.stake{value: 1 ether}();

        IOracle.SlashedValidator[] memory _validators = new IOracle.SlashedValidator[](2);
        _validators[0] = IOracle.SlashedValidator({publicKey: pubKeyValidator1, amount: 1 ether - 1});
        _validators[1] = IOracle.SlashedValidator({publicKey: pubKeyValidator2, amount: 0 ether});

        vm.prank(oracleMember1);
        oracleContract.slash(_validators);

        vm.prank(oracleMember1);
        issuerContract.stake{value: 10 ether}();

        assertEq(stkEth.balanceOf(user1), 1 ether);
        assertEq(stkEth.balanceOf(oracleMember1), 10 ether);
    }
```

Code Location:

Listing 6: Oracle.sol (Lines 219,228,235,236,238,239)

```
function slash(SlashedValidator[] memory _validators) external override whenNotPaused {
        require(isOracle(msg.sender), "Not oracle Member");
        bytes32 candidateId = keccak256(abi.encode(_validators));
        bytes32 voteId = keccak256(abi.encode(msg.sender, candidateId));
```

```
        require(!submittedVotes[voteId], "Oracles: already voted");

        submittedVotes[voteId] = true;
        uint256 candidateNewVotes = candidates[candidateId] + 1;
        candidates[candidateId] = candidateNewVotes;
        uint256 oracleMemberSize = oracleMemberLength();
        if (candidateNewVotes >= quorom) {
            // clean up votes
            int256 slashed_amount = 0;
            for (uint i = 0; i < _validators.length; ++i) {
                if (
                    IKeysManager(core().keysManager()).validators(_validators[i].publicKey).state ==
                    IKeysManager.State.ACTIVATED
                ) {
                    if (validatorSlashed[_validators[i].publicKey] != _validators[i].amount) {
                        slashed_amount += (int256(_validators[i].amount) -
                            int256(validatorSlashed[_validators[i].publicKey]));
                        validatorSlashed[_validators[i].publicKey] = _validators[i].amount;
                    }
                }
            }
            if (slashed_amount > 0) {
                withdrawals.slash(uint256(slashed_amount), true);
                changeCValue();
            } else if (slashed_amount < 0) {
                withdrawals.slash(uint256(int256(-1) * slashed_amount), false);
                changeCValue();
            }
            delete submittedVotes[voteId];

            for (uint256 i = 0; i < oracleMemberSize; i++) {
                delete submittedVotes[keccak256(abi.encode(oracleMembers.at(i), candidateId))];
            }
            delete candidates[candidateId];
        }
    }
```

BVSS:

AO:A/AC:L/AX:M/C:N/I:N/A:H/D:H/Y:N/R:N/S:U (6.3)

Recommendation:

It is recommended to have at least three oracle members on the protocol and set the quorum to at least two. As another recommendation, define a lower bound to maximum slashable amount such as MIN_SHARES to prevent under collateralization.

Remediation Plan:

SOLVED: The Persistence team solved this issue by migrating the slashing functionality to Oracle.pushData() function. The new function checks that if the slashing amount is higher than 1 ETH.

Commit ID: 424e44eb7d15e531487c60099962a6206ef27088

4.4 (HAL-04) LOSS OF FUNDS DUE TO MISSED LOGICAL IMPLEMENTATIONS - MEDIUM (5.6)

Description:

The StakingPool contract has a public StakingPool.updateRewardPerValidator() function to increase protocol rewards for validators. That function transfers the specified amount to the StakingPool contract, and it re-calculates accRewardPerValidator variable in every call.

Therefore, validators can claim rewards when someone calls the StakingPool.claimAndUpdateRewardDebt() function for them. However, rewards cannot be claimed at the first call of StakingPool.claimAndUpdateRewardDebt() function. The reason behind that problem is the pending uses user.amount variable, which shows the total of validators with DEPOSITED stage. However, that variable will always be zero at the first call since it is updated last. The pending will also return zero. That situation may cause a significant loss if someone tries to update validator rewards.

Additionally, there is no check in StakingPool.updateRewardPerValidator() to prevent token transfers when there are no validators with DEPOSITED stage. That can also cause loss of assets.

Proof of Concept:

Listing 7: Case1 - PoC

```
function test_claimAndUpdateTwiceLossOfRewardsCasePoC1() public {
        /*
        * case1:
        * 1. user2 updates reward per validator
        * 2. someone calls depositToEth2 to change validator's
↳ state
        * 3. user1 updates reward per validator
        * 4. someone calls claimAndUpdateRewardDebt function to
↳ grant rewards for node operator
        * 5. Loss of asset 10 Ether worth of stkEth tokens
        */
        vm.startPrank(user2);
        issuerContract.stake{value: 22 ether}();
        stkEth.approve(address(validatorPool), 10 ether);
        validatorPool.updateRewardPerValidator(10 ether);
        vm.stopPrank();

        vm.startPrank(user1);
        issuerContract.stake{value: 10 ether}();
        stkEth.approve(address(validatorPool), 10 ether);

        issuerContract.depositToEth2(pubKeyValidator1);
        validatorPool.updateRewardPerValidator(10 ether);
        validatorPool.claimAndUpdateRewardDebt(nodeOperator1);

        assertGt(stkEth.balanceOf(nodeOperator1), 0);
    }
```

FINDINGS & TECH DETAILS

Listing 8: Case2 - PoC

```
function test_claimAndUpdateTwiceLossOfRewardsCasePoC2() public {
        /*
        * case2:
        * 1. user2 updates reward per validator
        * 2. user1 calls depositToEth2 to change validator's state
        * 3. user1 calls claimAndUpdateRewardDebt function to grant rewards for node operator
        * 4. user1 updates reward per validator
        * 5. user1 calls claimAndUpdateRewardDebt function to grant rewards for node operator
        * 6. Loss of asset 15 Ether worth of stkEth tokens
        */
        vm.startPrank(user2);
        issuerContract.stake{value: 22 ether}();
        stkEth.approve(address(validatorPool), 10 ether);
        validatorPool.updateRewardPerValidator(10 ether);
        vm.stopPrank();

        vm.startPrank(user1);
        issuerContract.stake{value: 10 ether}();
        stkEth.approve(address(validatorPool), 10 ether);

        issuerContract.depositToEth2(pubKeyValidator1);
        validatorPool.claimAndUpdateRewardDebt(nodeOperator1);

        validatorPool.updateRewardPerValidator(10 ether);
        validatorPool.claimAndUpdateRewardDebt(nodeOperator1);

        assertGt(stkEth.balanceOf(nodeOperator1), 0);
    }
```

Code Location:

Listing 9: StakingPool.sol (Line 67)

```
function updateRewardPerValidator(uint256 newReward) public override {
        uint256 totalValidators = IOracle(core.oracle()).activatedValidators() +
            IIssuer(core.issuer()).pendingValidators();

```

```
        require(stkEth.transferFrom(_msgSender(), address(this),
↳ newReward), "Transfer failed");

        accRewardPerValidator += (newReward * 1e12) /
↳ totalValidators;
    }
```

Listing 10: StakingPool.sol (Lines 87,91,94,99)

```
86 function claimAndUpdateRewardDebt(address usr) external override {
87         UserInfo storage user = userInfos[usr];
88
89         uint256 userValidators = IKeysManager(core.keysManager()).
   ↳ nodeOperatorValidatorCount(usr);
90
91         uint256 pending = ((accRewardPerValidator * user.amount) /
   ↳ 1e12) - user.rewardDebt;
92
93         if (pending > 0) {
94             stkEth.transfer(usr, pending);
95             emit RewardRedeemed(pending, usr);
96         }
97
98         user.rewardDebt = (accRewardPerValidator * userValidators)
   ↳ / 1e12;
99         user.amount = userValidators;
100    }
```

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:M/Y:L/R:N/S:U (5.6)

Recommendation:

Consider adding a check for StakingPool.updateRewardPerValidator() function to prevent token transfers when there are no validators. Also, the pending formula should use user.amount as 1 for the first call of StakingPool.claimAndUpdateRewardDebt() if there is any validator with DEPOSITED or ACTIVE stage.

Remediation Plan:

RISK ACCEPTED: The risk of this finding was accepted by the Persistence team with the following reason:

> The rewards are only distributed for the keys which are deposited into consensus chain. This can only happen when there is enough ETH staked with us. This way, node operators are incentivized to submit and activate as many keys as possible.

4.5 (HAL-05) EXCESSIVELY CENTRALIZED FUNCTIONALITY - MEDIUM (5.0)

Description:

During the test, the Issuer.changePricePerShare() function on Optimism network currently was found excessively centralized. It has been seen that the address that deploys the StkEth contract on the Optimism Network can grant itself L2_MESSENGER authorization by calling the grantL2Messenger() function. Accounts with this authorization can also call the changePricePerShare() function. Although there is a low probability, the price per share value can be changed regardless of the StkEth token in the Ethereum network as a result of the compromise of the deployer address.

Therefore, it can lead to arbitrage opportunities.

Proof of Concept:

Listing 11: Centralized Function PoC

```
function test_canOwnerChangePrice() public {
        vm.startPrank(deployer);
        stkEth.grantL2Messenger(deployer);
        stkEth.changePricePerShare(1);
        vm.stopPrank();

        assertEq(stkEth.pricePerShare(), 1);
    }
```

Code Location:

Listing 12: L2-contracts/Optimism/contracts/StkEth.sol (Lines 64,65)

```
function changePricePerShare(
        uint256 _pricePerShare
    ) external override whenNotPaused onlyRole(L2_MESSENGER) {
        pricePerShare = _pricePerShare;
    }
```

BVSS:

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

Recommendation:

Consider using multi-sig wallet for the address which has L2_MESSENGER permission.

Remediation Plan:

PENDING: Multisig with time-lock will be added before mainnet release by the Persistence team. For further plan, Governance and Voting features will also be added.

4.6 (HAL-06) IMPROPER IMPLEMENTATION OF TRANSFERETHMAINNET FUNCTION - LOW (3.9)

Description:

There are two Issuer contracts on both Ethereum and Optimism networks. According to the developer's note, the Issuer.transferEthMainnet() function was designed to transfer the staked amount on L1 Ethereum. However, that function does not work as intended, and it directly deletes newEthStaked and newStkEthMinted information from the contract without sending any ETH.

Code Location:

Listing 13: L2-contracts/Optimism/contracts/Issuer.sol (Lines 51,52)

```
48 function transferEthMainnet() external override returns (bool) {
49         if (address(this).balance >= 0) {
50             //todo: insert sending to
51             newEthStaked = 0;
52             newStkEthMinted = 0;
53             return true;
54         } else {
55             return false;
56         }
57     }
```

BVSS:

AO:A/AC:L/AX:L/C:N/I:L/A:L/D:N/Y:N/R:N/S:C (3.9)

Recommendation:

Consider implementing a working version of transferEthMainnet() function. Also, consider removing that function entirely if it will not be used.

Remediation Plan:

SOLVED: The implementation of transferEthMainnet() function was corrected by the Persistence team. That function can transfer the Staked ETH amount to L1 Ethereum with the latest update.

Commit ID: 8fc5ac23c6c76cc2d9cb3359bea49bb00dfb4a96

4.7 (HAL-07) USE OF OPTIMISM TESTNET CHAIN ID - LOW (3.8)

Description:

During the assessment, it was seen that the OptimismMessenger contract uses 420 as the chain ID which belongs to the Optimism Testnet. As a result of deploying the OptimismMessenger contract with this value, it will cause a communication break between Ethereum and Optimism networks.

Code Location:

Listing 14: OptimismMessenger.sol (Line 15)

```
uint256 public constant destinationChainID = 420;
```

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:H/D:N/Y:N/R:P/S:U (3.8)

Recommendation:

Consider changing the chain ID to 10 which belongs to Optimism Mainnet.

Remediation Plan:

PENDING: The Persistence team was added a dev note to OptimismMessenger contract to change the chain ID to 10 right before the deployment.

4.8 (HAL-08) GETDEPOSITOPTIMISM FUNCTION ALWAYS GETS REVERTED - LOW (3.1)

Description:

During the tests, it was determined that the Issuer.getOptimismDeposit() function does not work correctly. The Issuer.getOptimismDeposit() function does not accept any function parameters. And, that function tries to decode the msg.data parameter as uint256. Additionally, no Role-based Access Control checks have been seen on this function. In a scenario where the function works correctly, the value of msg.data will be converted directly to uint256 without any checking, which may have unexpected results on the contract.

Code Location:

Listing 15: L1-contracts/contracts/IssuerUpgradable.sol (Line 188)

```
183 function getDepositOptimism() external payable {
184         //todo: put condition to check this data
185         if (msg.value > 0) {
186             //todo: get data and eth from lifi contract
187             ethStaked += msg.value;
188             stkEthMinted += abi.decode(msg.data, (uint256));
189         }
190     }
```

BVSS:

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

Recommendation:

Consider refactoring the `Issuer.getDepositOptimism()` since it does not work as intended.

Remediation Plan:

SOLVED: The `Issuer.getDepositOptimism()` function was refactored to `Issuer.getDepositL2()` function.

Commit ID: 8fc5ac23c6c76cc2d9cb3359bea49bb00dfb4a96

4.9 (HAL-09) USE OF TRANSFER/TRANSFERFROM METHOD INSTEAD OF SAFETRANSFER/SAFETRANSFERFROM - LOW (2.5)

Description:

It is considered a good practice to use a function like OpenZeppelin's safeTransfer/safeTransferFrom unless one is sure the given token reverts in case of a failed transfer. Some non-standard ERC20 tokens might cause silent failures of transfers and affect token accounting in contract.

Code Location:

Listing 16: StakingPool.sol (Line 67)

```
require(stkEth.transferFrom(_msgSender(), address(this), newReward
  ), "Transfer failed");
```

Listing 17: WithdrawalCredential.sol (Line 81)

```
core.stkEth().transfer(core.pstakeTreasury(), (pstakeRewards * 1
  e18) / pricePerShare);
```

BVSS:

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

Recommendation:

Consider using safeTransfer/safeTransferFrom consistently across the contracts.

Remediation Plan:

SOLVED: All transfer/transferFrom calls were replaced with safeTransfer /safeTransferFrom functions by the Persistence team.

Commit ID: 91c975d3fd48c37cabf160c4b206f4014b70f6e9

4.10 (HAL-10) IMPLEMENTATIONS CAN BE INITIALIZED - LOW (2.5)

Description:

The Issuer, StakingPool, WithdrawalCredential contracts are upgradable, inheriting from the Initializable contract. However, the current implementations are missing the _disableInitializers() function call in the constructors. Thus, an attacker can initialize the implementation. Usually, the initialized implementation has no direct impact on the proxy itself; however, it can be exploited in a phishing attack. In rare cases, the implementation might be mutable and may have an impact on the proxy.

BVSS:

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

Recommendation:

It is recommended to call _disableInitializers within the contract's constructor to prevent the implementation from being initialized.

Remediation Plan:

SOLVED: The Issuer, StakingPool, WithdrawalCredential contracts now implement the _disableInitializers() function call in the constructors.

Commit ID: ff40bb442aba920eb90542b9292cb66a8f6e3012

4.11 (HAL-11) FLOATING PRAGMA - INFORMATIONAL (0.0)

Description:

The project contains many instances of floating pragma. Contracts should be deployed with the same compiler version and flags that they have been tested with thoroughly. Locking the pragma helps to ensure that contracts do not accidentally get deployed using, for example, either an outdated compiler version that might introduce bugs that affect the contract system negatively or a pragma version too recent which has not been extensively tested.

Code Location:

All contracts are affected. (^0.8.0)

BVSS:

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

Recommendation:

Consider locking the pragma version with known bugs for the compiler version by removing the caret (^) symbol. When possible, do not use floating pragma in the final live deployment. Specifying a fixed compiler version ensures that the bytecode produced does not vary between builds. This is especially important if you rely on bytecode-level verification of the code.

Remediation Plan:

SOLVED: Pragma version was locked to 0.8.10 with the latest update.

Commit ID: 088bddb9d5349259e01870b1c1e2e6ccd5b4192a

4.12 (HAL-12) A MESSENGER CAN BE ADDED MORE THAN ONCE - INFORMATIONAL (0.0)

Description:

There is a function called addMessengers on the Oracle contract on the L1 side of the protocol. This function allows the owner to add a new messenger to the protocol. However, there is no check if a messenger already exists. As a result, it is possible to add the same messenger to the protocol multiple times.

Code Location:

Listing 18: L1-contracts/contracts/IssuerUpgradable.sol

```
function addMessenger(
        MessengerData[] calldata _messengers
    ) external onlyOwner returns (uint256[] memory) {
        if (_messengers.length == 0) revert ZeroMessengers();
        uint256[] memory messengerIds = new uint256[](_messengers.
↳ length);
        for (uint256 i = 0; i < _messengers.length; i++) {
            if (_messengers[i].messenger == address(0)) revert
↳ ZeroAddress();
            messengers.push(_messengers[i]);
            messengerIds[i] = messengers.length - 1;
            emit MessengerAdded(i, _messengers[i].messenger,
↳ _messengers[i].isEnabled);
        }
        return messengerIds;
    }
```

BVSS:

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

Recommendation:

Consider implementing a sanity check to control if the messenger was previously added to the protocol. As another solution, consider using `mapping` as `messengers` variable.

Remediation Plan:

PENDING: New Aggregator will be introduced in a future release. Therefore, multiple messenger logic will be removed.

4.13 (HAL-13) FOR LOOP OPTIMIZATIONS - INFORMATIONAL (0.0)

Description:

It has been observed all for loops in the protocol are not optimized. Suboptimal for loops can cost too much gas.

These for loops can be optimized with the suggestions above:

1. In Solidity (pragma 0.8.0 and later), adding the unchecked keyword for arithmetical operations can reduce gas usage on contracts where underflow/underflow is unrealistic. It is possible to save gas by using this keyword on multiple code locations.
2. In all for loops, the index variable is incremented using +=. It is known that, in loops, using ++i costs less gas per iteration than +=. This also affects incremented variables within the loop code block.
3. Do not initialize index variables with 0 Solidity already initializes these uint variables as zero.

Code Location:

All for loops are affected.

BVSS:

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 apply the following pattern for Solidity pragma version 0.8.0 and later.

FINDINGS & TECH DETAILS

Listing 19: Optimized For Loop structure

```
for (uint256 i; i < arrayLength; ) {
    . . .
    unchecked { ++i; }
```

Remediation Plan:

SOLVED: All for loops in the code base were optimized as suggested above by the Persistence team with the latest update.

Commit ID: 088bddb9d5349259e01870b1c1e2e6ccd5b4192a

4.14 (HAL-14) REDUNDANT LOGICS - INFORMATIONAL (0.0)

Description:

During the review performed, it was determined that there was more than one redundant logic on some contracts of the protocol.

Code Location:

Listing 20: KeysManager.sol (Line 77)

```
76 require(
77     IStakingPool(core().validatorPool()).numOfValidatorAllowed(
   ↳ validator.nodeOperator) >
78     nodeOperatorValidatorCount[validator.nodeOperator],
79     "KeysManager: validator deposit not added by node operator"
80 );
```

numOfValidatorAllowed function always returns type(uint256).max. So, there is no need to have that check since reaching to type(uint256).max as numOfValidatorAllowed is unrealistic.

Listing 21: Oracle.sol (Line 90)

```
89 function updateValidatorQuorom(uint32 latestQuorom) external
   ↳ onlyGovernor {
90         require(latestQuorom >= 0, "Quorom less that 0");
91         emit ValidatorQuoromUpdated(latestQuorom, validatorQuorom)
   ↳ ;
92         validatorQuorom = latestQuorom;
93     }
```

There is no need to use >= operator. The latestQuorom should be one at worst. Therefore, the condition of require function should be corrected as latestQuorom > 0 instead.

Listing 22: L1-contracts/contracts/token/StkEth.sol (Line 55)

```
49 function DOMAIN_SEPARATOR() public view returns (bytes32) {
50         uint256 chainId;
51         assembly {
52             chainId := chainid()
53         }
54         return
55             chainId == deploymentChainId ? _DOMAIN_SEPARATOR :
   ↳ _calculateDomainSeparator(chainId);
56     }
```

The _DOMAIN_SEPERATOR variable will always equal to _calculateDomainSeparator (chainId).

BVSS:

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

Recommendation:

Consider removing these redundant logics from contracts to increase readability and gas efficiency.

Remediation Plan:

SOLVED: All redundant logics were removed from the code base.

Commit ID: 088bddb9d5349259e01870b1c1e2e6ccd5b4192a

4.15 (HAL-15) UNUSED IMPORTS, VARIABLES AND FUNCTIONS - INFORMATIONAL (0.0)

Description:

There are numerous unused imports, variables, and functions on the code base. These variables should be cleaned up from the code if they have no purpose. Clearing these variables can reduce gas usage during the deployment of contracts.

Code Location:

Listing 23: Unused imports, variables and functions

```
L1-contracts/contracts/Core.sol#L8 - unused import
L1-contracts/contracts/CoreRef.sol#L8 - unused import
L1-contracts/contracts/IssuerUpgradable.sol#L9,L10 - unused import
L1-contracts/contracts/KeysManager.sol#L14,L15 - unused variables
L1-contracts/contracts/Oracle.sol#L15,L17 - unused variables
L1-contracts/contracts/StakingPool.sol#L103 - deprecated function
```

BVSS:

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

Recommendation:

Consider removing unused imports, variables, and functions from the code.

Remediation Plan:

SOLVED: Unused imports, variables, and functions were removed from the code base.

Commit ID: 088bddb9d5349259e01870b1c1e2e6ccd5b4192a

4.16 (HAL-16) OPEN TODOS - INFORMATIONAL (0.0)

Description:

Open To-dos can hint at programming or architectural errors that still need to be fixed.

Code Location:

Listing 24: Open TODOs

```
L1-contracts/contracts/IssuerUpgradable.sol#L184
L1-contracts/contracts/IssuerUpgradable.sol#L186
StakingPool.sol#L102
WithdrawalCredential.sol#L108
L2-contracts/Optimism/contracts/Issuer.sol#L50
```

BVSS:

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 resolve all TODOs before the prod stage.

Remediation Plan:

SOLVED: Open TODOs were resolved from the code base by the Persistence team.

Commit ID: 088bddb9d5349259e01870b1c1e2e6ccd5b4192a

4.17 (HAL-17) STRICTLY PACKED VARIABLES CONSUMES LESS GAS - INFORMATIONAL (0.0)

Description:

In Ethereum, storage operates as a key-value repository, where both keys and values occupy 32-byte spaces. Upon storage allocation, all variables with static sizes (excluding mappings and dynamically-sized arrays) are sequentially written in the order of their declaration, beginning at position 0. Frequently utilized data types like bytes32, uint, and int occupy precisely one 32-byte slot in storage. This approach outlines a method to optimize gas consumption by utilizing smaller data types (such as bytes16 or uint32) whenever possible. By doing so, the EVM can consolidate them into a single 32-byte slot, resulting in reduced storage usage and gas savings.

Code Location:

Listing 25: L1-contracts/contracts/token/StkEth.sol

```
bytes32 public constant PERMIT_TYPEHASH =
        0x6e71edae12b1b97f4d1f60370fef10105fa2faae0126114a169c64845d6126c9;

    mapping(address => uint256) public nonces;

    event burnToken(address user, uint256 amount);

    // solhint-disable-next-line var-name-mixedcase
    bytes32 private immutable _DOMAIN_SEPARATOR;

    uint256 public immutable deploymentChainId;
```

FINDINGS & TECH DETAILS

Listing 26: Oracle.sol

```
    uint128 internal constant ETH2_DENOMINATION = 1e9;
    uint256 constant BASIS_POINT = 10000;
    uint256 public DEPOSIT_LIMIT = 32e18;

    mapping(bytes => uint256) public validatorSlashed;
    uint256 lastValidatorActivation;
    uint32 quorom;
    uint32 validatorQuorom;
    uint256 public override activatedValidators = 1;
    uint32 pStakeCommission;
    uint32 valCommission;
    IWithdrawalCredential public withdrawals;
    uint64 public activateValidatorDuration = 10 minutes;
```

BVSS:

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 always recommended to strictly tight storage slots to save gas. You can perform the changes as in the following example:

Listing 27: Unpacked storage slots

```
    uint128 internal constant ETH2_DENOMINATION = 1e9;
    uint256 constant BASIS_POINT = 10000;
    uint256 public DEPOSIT_LIMIT = 32e18;

    mapping(bytes => uint256) public validatorSlashed;
    uint256 lastValidatorActivation;
    uint32 quorom;
    uint32 validatorQuorom;
    uint256 public activatedValidators = 1;
    uint32 pStakeCommission;
    uint32 valCommission;
    //IWithdrawalCredential public withdrawals;
    uint64 public activateValidatorDuration = 10 minutes;

```

```
//execution cost: 226431
```

Listing 28: Strictly packed storage slots

```
    uint256 constant BASIS_POINT = 10000;
    uint256 public DEPOSIT_LIMIT = 32e18;
    uint256 lastValidatorActivation;
    uint256 public activatedValidators = 1;
    uint128 internal constant ETH2_DENOMINATION = 1e9;
    uint64 public activateValidatorDuration = 10 minutes;
    uint32 quorom;
    uint32 validatorQuorom;
    uint32 pStakeCommission;
    uint32 valCommission;
    mapping(bytes => uint256) public validatorSlashed;
    //IWithdrawalCredential public withdrawals;

//execution cost: 226381
```

It is also important to introduce constant and immutable keywords to increase gas efficiency for unchangeable variables.

Remediation Plan:

SOLVED: The Persistence team solved this issue in commit 91c975d3fd48c37cabf160c4b206f4014b70f6e9. Contract variables are now packed for the Oracle contract.

AUTOMATED TESTING

5.1 STATIC ANALYSIS REPORT

Description:

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' ABIs across the entire code-base.

Results:

As a result of the tests carried out with the Slither tool, some results were obtained and these results were reviewed by Halborn. Based on the results reviewed, most of these vulnerabilities were determined to be false positives and these results were not included in the report.

5.2 AUTOMATED SECURITY SCAN

Description:

Halborn used automated security scanners to assist with detection of well-known security issues and to identify low-hanging fruits on the targets for this engagement. Among the tools used was MythX, a security analysis service for Ethereum smart contracts. MythX performed a scan on the smart contracts and sent the compiled results to the analyzers in order to locate any vulnerabilities.

Results:

The findings obtained as a result of the MythX scan were examined, and the findings were not included in the report because they were found to be false positive.

THANK YOU FOR CHOOSING

HALBORN