HALBORN

Stader Labs - ERC20 Staking

Smart Contract Security Audit

Prepared by: **Halborn**

Date of Engagement: **July 26th, 2022 - August 4th, 2022**

Visit: **Halborn.com**

DOCUMENT REVISION HISTORY

VERSION	MODIFICATION	DATE	AUTHOR
0.1	Document Creation	08/02/2022	Gustavo Dutra
0.2	Document Updates	08/03/2022	Mostafa Yassine
0.2	Document Updates	08/03/2022	Manuel Garcia
0.3	Document Updates	08/04/2022	Gustavo Dutra
0.4	Draft Review	08/04/2022	Kubilay Onur Gungor
0.5	Draft Review	08/05/2022	Gabi Urrutia
1.0	Remediation Plan	08/12/2022	Gustavo Dutra
1.1	Remediation Plan Review	08/15/2022	Gabi Urrutia
1.2	Document Updates	09/05/2022	Manuel García

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
Kubilay Onur Gungor	Halborn	Kubilay.Gungor@halborn.com
Gustavo Dutra	Halborn	Gustavo.Dutra@halborn.com
Manuel García	Halborn	Manuel.Diaz@halborn.com

EXECUTIVE OVERVIEW

1.1 INTRODUCTION

Stader Labs engaged Halborn to conduct a security audit on their smart contracts beginning on July 26th, 2022 and ending on August 04th, 2022. The security assessment was scoped to the smart contracts provided in the GitHub repositories stader-labs/sd-erc20-staking-v1

1.2 AUDIT SUMMARY

The team at Halborn was provided a week for the engagement and assigned three full-time security engineers to audit the security of the smart contract. The security engineers are blockchain and smart-contract security experts with advanced penetration testing, smart-contract hacking, and deep knowledge of multiple blockchain protocols.

The purpose of this audit 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 addressed by Stader Labs team.

POST ASSESSMENT:

After the initial assessment the commit cd4518ee9bd31718c53a8dc27625a3b48a7d8681 was also analysed to ensure minor changes in the code were secured.

1.3 TEST APPROACH & METHODOLOGY

Halborn performed a combination of manual and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard

to the scope of this audit. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance coverage of the code and can quickly identify items that do not follow the security best practices. The following phases and associated tools were used during the audit:

- 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
- Manual testing by custom scripts
- Scanning of solidity files for vulnerabilities, security hotspots or bugs. (MythX)
- Static Analysis of security for scoped contract, and imported functions. (Slither)
- Testnet deployment (Brownie, Remix IDE)

RISK METHODOLOGY:

Vulnerabilities or issues observed by Halborn are ranked based on the risk assessment methodology by measuring the **LIKELIHOOD** of a security incident and the **IMPACT** should an incident occur. This framework works for communicating the characteristics and impacts of technology vulnerabilities. The quantitative model ensures repeatable and accurate measurement while enabling users to see the underlying vulnerability characteristics that were used to generate the Risk scores. For every vulnerability, a risk level will be calculated on a scale of 5 to 1 with 5 being the highest likelihood or impact.

RISK SCALE - LIKELIHOOD

5 - Almost certain an incident will occur.
4 - High probability of an incident occurring.
3 - Potential of a security incident in the long term.

2 - Low probability of an incident occurring.
1 - Very unlikely issue will cause an incident.

RISK SCALE - IMPACT

5 - May cause devastating and unrecoverable impact or loss.
4 - May cause a significant level of impact or loss.
3 - May cause a partial impact or loss to many.
2 - May cause temporary impact or loss.
1 - May cause minimal or un-noticeable impact.

The risk level is then calculated using a sum of these two values, creating a value of 10 to 1 with 10 being the highest level of security risk.

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL

10 - CRITICAL
9 - 8 - HIGH
7 - 6 - MEDIUM
5 - 4 - LOW
3 - 1 - VERY LOW AND INFORMATIONAL

1.4 SCOPE

IN-SCOPE:

The security assessment was scoped to the following smart contracts:

- Ownable.sol
- Rewards.sol
- Staking.sol
- Timelock.sol
- Undelegation.sol
- XSD.sol

Commit ID:

- 933bdbc97988639f42995537ea6716d7ee646aba

Fixed commit IDs:

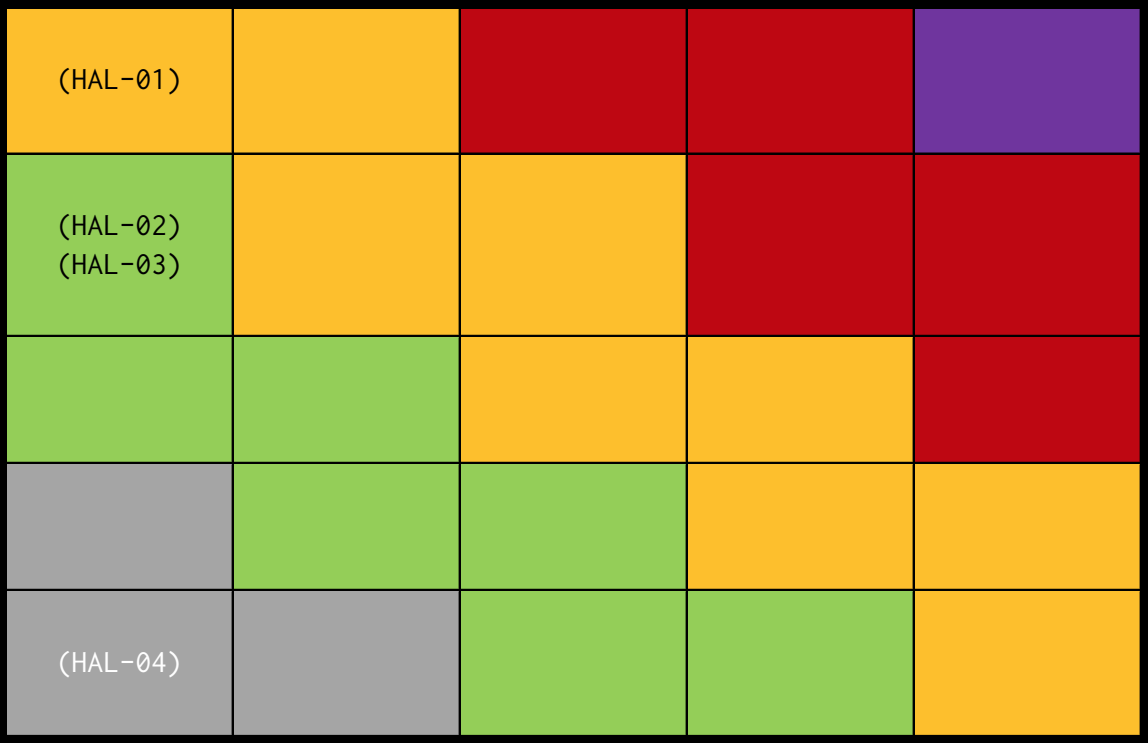
- d4463346158a014a95f699a07b761770dff61515
- e600a0a08719e3140ed955d6dd316bdf606bdeb1
- cd4518ee9bd31718c53a8dc27625a3b48a7d8681

2. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	0	1	2	1

LIKELIHOOD

IMPACT



EXECUTIVE OVERVIEW

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
HAL01 - OWNER CAN POTENTIALLY WITHDRAW ALL THE STAKING FUNDS	Medium	SOLVED - 08/11/2022
HAL02 - MAX STAKING AMOUNT CAN BE SET TO ZERO	Low	SOLVED - 08/11/2022
HAL03 - MIN STAKING AMOUNT CAN BE GREATER THAN THE MAX STAKING AMOUNT	Low	SOLVED - 08/11/2022
HAL04 - REWARDS EMISSION RATE CAN BE SET TO ZERO	Informational	SOLVED - 08/11/2022

FINDINGS & TECH DETAILS

3.1 (HAL-01) OWNER CAN POTENTIALLY WITHDRAW ALL THE STAKING FUNDS - MEDIUM

Description:

The function queuePartialFunds, that is inherited from the Timelock contract in Staking.sol enables the owner to queue to withdraw all the funds of the Staking contract after the lockedPeriod passes.

However, the owner can also change the lockedPeriod to zero, enabling instant withdrawal after calling the queuePartialFunds function.

This causes a centralization issue, leaving space for more potential damage in case the owner's account gets compromised somehow.

Proof of Concept:

We can see the following scenario in this Proof of Concept:

1. user1 and user2 staked 1000 Stader tokens;
2. The compromised admin sets lockedPeriod to zero;
3. Compromised admin queue for all funds to withdraw;
4. Withdraw it;
5. Compromised admin gets the balance from the withdrawal;
6. user1 cannot unstake their funds anymore.

```
<Transaction '0xfb57b38a90e258985c9055ce8b1e                de474d7c1354724275a59f8aed4b65c2'>
>>> staking.stake(1000, {'from': user2})
Transaction sent: 0xcdc3624bff3c402f74603da4600953b0a98cfe0bc8179dc06861833fef96adb9
  Gas price: 0.0 gwei   Gas limit: 12000000   Nonce: 1
  Staking.stake confirmed   Block: 18   Gas used: 86255 (0.72%)

<Transaction '0xcdc3624bff3c402f74603da4600953b0a98cfe0bc8179dc06861833fef96adb9'>
>>> xsd.balanceOf(user1)
1000
>>> xsd.balanceOf(user2)
1000
>>> staking.setLockedPeriod(0, {'from':admin})
Transaction sent: 0x1052fbd0c68dd2dc7f2db98237dcf422f980cc8c0bc8d200a157b6afe4de5764
  Gas price: 0.0 gwei   Gas limit: 12000000   Nonce: 10
  Staking.setLockedPeriod confirmed   Block: 19   Gas used: 15311 (0.13%)

<Transaction '0x1052fbd0c68dd2dc7f2db98237dcf422f          c8c0bc8d200a157b6afe4de5764'>
>>> staking.queueAllFunds(admin, {'from':admin})
Transaction sent: 0x527880aa2e9b5691a1e4d6a4e41c80d9205c9b86e92ccc07623d05c8eb516a70
  Gas price: 0.0 gwei   Gas limit: 12000000   Nonce: 11
  Staking.queueAllFunds confirmed   Block: 20   Gas used: 114303 (0.95%)

<Transaction '0x527880aa2e9b5691a1e4d6a4e41c80d9205c9b86e92ccc07623d05c8eb516a70'>
>>> before_balance = stader.balanceOf(admin)
>>> staking.withdraw(0, {'from':admin})
Transaction sent: 0xfdef21ef92900e56f6bf73d53d9df86081e1fdaca32429163bdd2296be766248
  Gas price: 0.0 gwei   Gas limit: 12000000   Nonce: 12
  Staking.withdraw confirmed   Block: 21   Gas used: 37655 (0.31%)

<Transaction '0xfdef21ef92900e56f6bf73d53d9df86081e1fdaca32429163bdd2296be766248'>
>>> staking.withdraw(0, {'from':admin})
Transaction sent: 0x0e2f797838448517b82ce2fa12d2064c7177341d7334b756d79fab16ae833469
  Gas price: 0.0 gwei   Gas limit: 12000000   Nonce: 13
  Staking.withdraw confirmed (No funds to withdraw)   Block: 22   Gas used: 31229 (0.26%)

<Transaction '0x0e2f797838448517b82ce2fa12d2064c7177341d7334b756d79fab16ae833469'>
>>> staking.withdraw(1, {'from':admin})
Transaction sent: 0x7923a02796eb380187a0ed2557a0f5a54c12ff8728c70aea294bc6e77d84d24a
  Gas price: 0.0 gwei   Gas limit: 12000000   Nonce: 14
  Staking.withdraw confirmed (No funds to withdraw)   Block: 23   Gas used: 31241 (0.26%)

<Transaction '0x7923a02796eb380187a0ed2557a0f5a54c12ff8728c70aea294bc6e77d84d24a'>
>>> stader.balanceOf(admin)
1499999999999999999988002000
>>> stader.balanceOf(admin) - before_balance
2000
>>> staking.unstake(1000, {'from': user1
... })
Transaction sent: 0x62d7911a6aa269df68b306a79389b2e46f736e57e9db9b27ec96469af13fc23c
  Gas price: 0.0 gwei   Gas limit: 12000000   Nonce: 6
  Staking.unstake confirmed (ERC20: insufficient allowance)   Block: 24   Gas used: 40680 (0.34%)

<Transaction '0x62d7911a6aa269df68b306a79389b2e46f736e57e9db9b27ec96469af13fc23c'>
>>> stader.balanceOf(staking)
0
>>>
```

1
2
3
4
5
6

Risk Level:

Likelihood - 1

Impact - 5

Recommendation:

1. Set a minimum limit for the lockedPeriod, so there is no chance of an instant withdraw from the queue.
2. Consider removing the queuePartialFunds function.

Remediation Plan:

SOLVED: The Stader Labs team fixed the issue by setting a fixedLockedPeriod of 1 day, which will be the minimum lock period that can be set in the contract. With this, even if the owner was compromised, they would have time to alert users to withdraw their funds.

3.2 (HAL-02) MAX STAKING AMOUNT CAN BE SET TO ZERO - LOW

Description:

The function updateMaxDeposit accepts an uint256 value _newMaxDeposit of 0. This will prevent users from being able to stake any amount, as the check in the stake function requires the staked amount to be smaller than the maxDeposit amount.

This could be done accidentally by the owner of the contract or in case the owner gets compromised.

Code Location:

Listing 1: staking.sol (Line 159)

```
157   /// @notice Set maximum deposit amount (onlyOwner)
158   /// @param _newMaxDeposit the maximum deposit amount in
 ↳ multiples of 10**8
159   function updateMaxDeposit(uint256 _newMaxDeposit) external
 ↳ onlyOwner {
160     require(maxDeposit != _newMaxDeposit, 'Max Deposit is
 ↳ unchanged');
161     emit maxDepositChanged(_newMaxDeposit, maxDeposit);
162     maxDeposit = _newMaxDeposit;
163   }
```

Listing 2: staking.sol (Line 71)

```
67  function stake(uint256 _amount) external whenNotPaused
 ↳ nonReentrant {
68     require(!isStakePaused, 'Staking is paused');
69
70     require(
71       _amount > minDeposit && _amount <= maxDeposit,
72       'Deposit amount must be within valid range'
73     );
74     require(
```

```
75        staderToken.balanceOf(address(rewardsContractAddress)) > 0,
76        'Rewards contract cannot have zero balance'
77      );
```

Risk Level:

Likelihood - 1
Impact - 4

Recommendation:

Verify that the _newMaxDeposit received in the function updateMaxDeposit exceeds zero.

Remediation Plan:

SOLVED: The Stader Labs team fixed the issue by checking if the received _newMaxDeposit exceeds zero.

3.3 (HAL-03) MIN STAKING AMOUNT CAN BE GREATER THAN THE MAX STAKING AMOUNT - LOW

Description:

The function updateMinDeposit accepts a uint256 value _newMinDeposit, but it does not verify that this value is smaller than the maxDeposit amount. In case the minDeposit is greater than the maxDeposit, then the required statement in the staking function will always fail.

This could be done accidentally by the owner of the contract or in case the owner gets compromised.

Code Location:

Listing 3: staking.sol

```
151  /// @notice Set minimum deposit amount (onlyOwner)
152   /// @param _newMinDeposit the minimum deposit amount in
 ↳ multiples of 10**8
153   function updateMinDeposit(uint256 _newMinDeposit) external
 ↳ onlyOwner {
154     require(minDeposit != _newMinDeposit, 'Min Deposit is
 ↳ unchanged');
155     emit minDepositChanged(_newMinDeposit, minDeposit);
156     minDeposit = _newMinDeposit;
157   }
```

Listing 4: staking.sol (Line 71)

```
67  function stake(uint256 _amount) external whenNotPaused
 ↳ nonReentrant {
68     require(!isStakePaused, 'Staking is paused');
69
70     require(
71       _amount > minDeposit && _amount <= maxDeposit,
72       'Deposit amount must be within valid range'
```

```
73      );
74      require(
75        staderToken.balanceOf(address(rewardsContractAddress)) > 0,
76        'Rewards contract cannot have zero balance'
77      );
```

Risk Level:

Likelihood - 1
Impact - 4

Recommendation:

Verify that _newMinDeposit is strictly smaller than maxDeposit.

Remediation Plan:

SOLVED: The Stader Labs team fixed the issue by checking if the value received in updateMinDeposit is less than maxDeposit.

3.4 (HAL-04) REWARDS EMISSION RATE CAN BE SET TO ZERO - INFORMATIONAL

Description:

In the contract Rewards, the state variable emissionRate is used to calculate the rewards' distribution in the function distributeStakingRewards().

This state variable is set by the function setEmissionRate(), which does not check if the received value is different from zero.

If this value is set to zero, the calculation of the rewards' distribution will always result in zero; therefore, no reward will be distributed.

This can happen mistakenly or in case the owner gets compromised.

Code Location:

Listing 5: Rewards.sol (Line 70)

```
63   /// @dev currently we will distribute the rewards every 24 hours
↳  and is controlled by offchain function
64   function distributeStakingRewards() external whenNotPaused
↳ nonReentrant {
65     require(staderToken.balanceOf(address(this)) > 0, 'Contract
↳ balance should be greater than 0');
66     uint256 currentTimestamp = block.timestamp;
67     uint256 epochDelta = (currentTimestamp - lastRedeemedTimestamp
↳ );
68     lastRedeemedTimestamp = currentTimestamp;
69     epoch++;
70     uint256 epochRewards = (epochDelta * emissionRate);
71
72     uint256 totalRewards = staderToken.balanceOf(address(this));
73     if (epochRewards > totalRewards) epochRewards = totalRewards;
↳ // this is important
74     emit DistributedRewards(stakingContractAddress, epochRewards,
↳ currentTimestamp);
```

```
75      require(
76        staderToken.transfer(stakingContractAddress, epochRewards),
77        'Failed to transfer rewards'
78      );
79    }
```

Listing 6: Rewards.sol (Line 89)

```
85    /// @notice Emission rate is defined by SD per second.
86    /// @param _emissionRate new value for the emission rate
87    function setEmissionRate(uint256 _emissionRate) external
↳ onlyOwner {
88      require(emissionRate != _emissionRate, 'Emission rate
↳ unchanged');
89      emissionRate = _emissionRate;
90      emit NewEmissionRate(emissionRate);
91    }
```

Risk Level:

Likelihood - 1
Impact - 1

Recommendation:

Check if the received value in setEmissionRate is different from zero, and revert the transaction otherwise.

Remediation Plan:

SOLVED: The Stader Labs team fixed the issue by checking to only accept a value greater than zero in the setEmissionRate function.

AUTOMATED TESTING

4.1 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' APIs across the entire code-base.

Slither results:

Ownable.sol

```
Context._msgData() (../node_modules/@openzeppelin/contracts/utils/Context.sol#21-23) is never used and should be removed
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#dead-code

Pragma version^0.8.0 (../node_modules/@openzeppelin/contracts/utils/Context.sol#4) allows old versions
Pragma version^0.8.0 (Ownable.sol#4) allows old versions
solc-0.8.9 is not recommended for deployment
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity
```

Timelock.sol

```
Timelock.withdraw(uint256) (Timelock.sol#95-107) uses a dangerous strict equality:
        - staderToken.balanceOf(address(this)) == 0 (Timelock.sol#96)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#dangerous-strict-equalities

Timelock.withdraw(uint256) (Timelock.sol#95-107) uses timestamp for comparisons
        Dangerous comparisons:
        - index >= withdrawQueue.length (Timelock.sol#97)
        - withdrawData.timestamp + lockedPeriod >= block.timestamp (Timelock.sol#99)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#block-timestamp
```

Undelegation.sol

```
Undelegation.withdraw(uint256) (Undelegation.sol#65-77) uses timestamp for comparisons
        Dangerous comparisons:
        - require(bool,string)(undelegateData.timestamp + unbondingTime <= block.timestamp,Release time not reached) (Undelegation.sol#68-71)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#block-timestamp
```

Rewards.sol

```
Contract locking ether found:
        Contract Rewards (Rewards.sol#13-143) has payable functions:
         - Rewards.fallback() (Rewards.sol#135-137)
         - Rewards.receive() (Rewards.sol#140-142)
        But does not have a function to withdraw the ether
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#contracts-that-lock-ether

Rewards.distributeStakingRewards() (Rewards.sol#64-79) uses timestamp for comparisons
        Dangerous comparisons:
        - epochRewards > totalRewards (Rewards.sol#73)
        - require(bool,string)(staderToken.transfer(stakingContractAddress,epochRewards),Failed to transfer rewards) (Rewards.sol#75-78)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#block-timestamp

Pragma version^0.8.0 (../node_modules/@openzeppelin/contracts/security/Pausable.sol#4) allows old versions
Pragma version^0.8.0 (../node_modules/@openzeppelin/contracts/security/ReentrancyGuard.sol#4) allows old versions
Pragma version^0.8.0 (../node_modules/@openzeppelin/contracts/token/ERC20/IERC20.sol#4) allows old versions
Pragma version^0.8.1 (../node_modules/@openzeppelin/contracts/utils/Address.sol#4) allows old versions
Pragma version^0.8.0 (../node_modules/@openzeppelin/contracts/utils/Context.sol#4) allows old versions
Pragma version^0.8.0 (Ownable.sol#4) allows old versions
Pragma version0.8.9 (Rewards.sol#2) necessitates a version too recent to be trusted. Consider deploying with 0.6.12/0.7.6/0.8.7
solc-0.8.9 is not recommended for deployment
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity

Low level call in Address.sendValue(address,uint256) (../node_modules/@openzeppelin/contracts/utils/Address.sol#60-65):
        - (success) = recipient.call{value: amount}() (../node_modules/@openzeppelin/contracts/utils/Address.sol#63)
Low level call in Address.functionCallWithValue(address,bytes,uint256,string) (../node_modules/@openzeppelin/contracts/utils/Address.sol#128-139):
        - (success,returndata) = target.call{value: value}(data) (../node_modules/@openzeppelin/contracts/utils/Address.sol#137)
Low level call in Address.functionStaticCall(address,bytes,string) (../node_modules/@openzeppelin/contracts/utils/Address.sol#157-166):
        - (success,returndata) = target.staticcall(data) (../node_modules/@openzeppelin/contracts/utils/Address.sol#164)
Low level call in Address.functionDelegateCall(address,bytes,string) (../node_modules/@openzeppelin/contracts/utils/Address.sol#184-193):
        - (success,returndata) = target.delegatecall(data) (../node_modules/@openzeppelin/contracts/utils/Address.sol#191)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#low-level-calls

Parameter Rewards.setEmissionRate(uint256)._emissionRate (Rewards.sol#87) is not in mixedCase
Parameter Rewards.setStakingContractAddress(address)._stakingContractAddress (Rewards.sol#95) is not in mixedCase
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#conformance-to-solidity-naming-conventions
```

AUTOMATED TESTING

Staking.sol

```
Contract locking ether found:
        Contract Staking (Staking.sol#15-215) has payable functions:
         - Staking.fallback() (Staking.sol#207-209)
         - Staking.receive() (Staking.sol#212-214)
        But does not have a function to withdraw the ether
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#contracts-that-lock-ether

Reentrancy in Staking.stake(uint256) (Staking.sol#67-100):
        External calls:
        - xStaderToken.mint(msg.sender,amountToSend) (Staking.sol#87)
        - xStaderToken.mint(msg.sender,amountToSend) (Staking.sol#92)
        Event emitted after the call(s):
        - Staked(msg.sender,_amount,amountToSend) (Staking.sol#95)
Reentrancy in Staking.unstake(uint256) (Staking.sol#103-120):
        External calls:
        - require(bool,string)(xStaderToken.transferFrom(msg.sender,address(this),_share),Failed to transfer xSD) (Staking.sol#109)
        - xStaderToken.burn(_share) (Staking.sol#110)
        Event emitted after the call(s):
        - UnStaked(msg.sender,sdToSend,_share) (Staking.sol#112)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-3


Variable Staking.constructor(IERC20,XSD,address,address)._staderToken (Staking.sol#53) is too similar to Staking.xStaderToken (Staking.sol#16)
Variable Timelock.constructor(IERC20,address)._staderToken (Timelock.sol#53) is too similar to Staking.xStaderToken (Staking.sol#16)
Variable XSD.SUPPLY_ROLE (XSD.sol#12) is too similar to XSD.setSupplyRole(address)._supplyRole (XSD.sol#31)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#variable-names-are-too-similar

Staking.slitherConstructorVariables() (Staking.sol#15-215) uses literals with too many digits:
        - maxDeposit = 1000000 * 10 ** 18 (Staking.sol#22)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#too-many-digits
```

- No major issues found by Slither.

4.2 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.

MythX results:

SD.sol

Report for SD.sol
https://dashboard.mythx.io/#/console/analyses/6020c0be-a497-4bfb-b5f3-db12c1186f6a

Line	SWC Title	Severity	Short Description
2	(SWC-103) Floating Pragma	Low	A floating pragma is set.

Rewards.sol

Report for Rewards.sol
https://dashboard.mythx.io/#/console/analyses/0a5ca6fb-633b-4991-ba2c-e9065f57b565

Line	SWC Title	Severity	Short Description
25	(SWC-108) State Variable Default Visibility	Low	State variable visibility is not set.
67	(SWC-101) Integer Overflow and Underflow	Unknown	Arithmetic operation "-" discovered
69	(SWC-101) Integer Overflow and Underflow	Unknown	Arithmetic operation "++" discovered
70	(SWC-101) Integer Overflow and Underflow	Unknown	Arithmetic operation "*" discovered

Staking.sol

Report for Rewards.sol
https://dashboard.mythx.io/#/console/analyses/0a5ca6fb-633b-4991-ba2c-e9065f57b565

Line	SWC Title	Severity	Short Description
25	(SWC-108) State Variable Default Visibility	Low	State variable visibility is not set.
67	(SWC-101) Integer Overflow and Underflow	Unknown	Arithmetic operation "-" discovered
69	(SWC-101) Integer Overflow and Underflow	Unknown	Arithmetic operation "++" discovered
70	(SWC-101) Integer Overflow and Underflow	Unknown	Arithmetic operation "*" discovered

Timelock.sol

- No major issues were found by MythX. MythX correctly flagged that some state variables are missing the public/private keyword, so all of them will be declared as private by default.

THANK YOU FOR CHOOSING

// HALBORN