Crescite Staking and Escapable Contracts

Smart Contract Security Audit

Prepared by ShellBoxes

September 19th, 2023 – September 22nd, 2023

Shellboxes.com

contact@shellboxes.com

Document Properties

Client	Crescite
Version	1.0
Classification	Public

Scope

Repository	Commit Hash
https://github.com/Crescite/crescite-token	55e78f2e69eea6a65516c05e87a5f51c9399b3cb

Re-Audit

Repository	Commit Hash
https://github.com/Crescite/crescite-token	be5f9121eb5432ef442ddf0a23c172a5280d3552

Contacts

COMPANY	EMAIL
ShellBoxes	contact@shellboxes.com

Contents

1 Introduction

Crescite engaged ShellBoxes to conduct a security assessment on the Crescite Staking and Escapable Contracts beginning on September 19th, 2023 and ending September 22nd, 2023. In this report, we detail our methodical approach to evaluate potential security issues associated with the implementation of smart contracts, by exposing possible semantic discrepancies between the smart contract code and design document, and by recommending additional ideas to optimize the existing code. Our findings indicate that the current version of smart contracts can still be enhanced further due to the presence of many security and performance concerns.

This document summarizes the findings of our audit.

1.1 About Crescite

The Crescite Protocol is a Layer 2 built on the open-source XDC Network, allowing for a decentralized, hybrid, interoperable, and liquid network.The Crescite Token is an XRC-20 Token. The token is meant to be used for functional utility within their community platform to grow the ideals of a faith-based blockchain community and further ESG impact across the world.

Issuer	Crescite
Website	`https://www.crescite.org`
Type	Solidity Smart Contract
Whitepaper	`https://www.crescite.org/_files/ugd/fd64a1_eeca34ad5f6e4bb68701fceb9b7aeac7.pdf`
Audit Method	Whitebox

1.2 Approach & Methodology

ShellBoxes used a combination of manual and automated security testing to achieve a balance between efficiency, timeliness, practicability, and correctness within the audit's scope. While manual testing is advised for identifying problems in logic, procedure, and implementation, automated testing techniques help to expand the coverage of smart contracts and can quickly detect code that does not comply with security best practices.

1.2.1 Risk Methodology

Vulnerabilities or bugs identified by ShellBoxes are ranked using a risk assessment technique that considers both the LIKELIHOOD and IMPACT of a security incident. This framework is effective at conveying the features and consequences of technological vulnerabilities.

Its quantitative paradigm enables repeatable and precise measurement, while also revealing the underlying susceptibility characteristics that were used to calculate the Risk scores. A risk level will be assigned to each vulnerability on a scale of 5 to 1, with 5 indicating the greatest possibility or impact.

- Likelihood quantifies the probability of a certain vulnerability being discovered and exploited in the untamed.
- Impact quantifies the technical and economic costs of a successful attack.
- Severity indicates the risk's overall criticality.

Probability and impact are classified into three categories: H, M, and L, which correspond to high, medium, and low, respectively. Severity is determined by probability and impact and is categorized into four levels, namely Critical, High, Medium, and Low.

Impact				
	High	Critical	High	Medium
	Medium	High	Medium	Low
	Low	Medium	Low	Low
		High	Medium	Low

Likelihood

2 Findings Overview

2.1 Summary

The following is a synopsis of our conclusions from our analysis of the Crescite Staking and Escapable Contracts implementation. During the first part of our audit, we examine the smart contract source code and run the codebase via a static code analyzer. The objective here is to find known coding problems statically and then manually check (reject or confirm) issues highlighted by the tool. Additionally, we check business logics, system processes, and DeFi-related components manually to identify potential hazards and/or defects.

2.2 Key Findings

In general, these smart contracts are well-designed and constructed, but their implementation might be improved by addressing the discovered flaws, which include 1 critical-severity, 1 high-severity, 3 medium-severity, 5 low-severity vulnerabilities.

Vulnerabilities	Severity	Status
SHB.1. Missing Storage Gaps in StakingUpgradeable Contract	CRITICAL	Fixed
SHB.2. Insufficient Funds for Staking Rewards	HIGH	Mitigated
SHB.3. Precision Loss in Reward Calculation	MEDIUM	Fixed
SHB.4. Potential DoS Due to Unoptimized Position Removal	MEDIUM	Mitigated
SHB.5. Excessive Power to Owner and _escapeHatchCaller	MEDIUM	Acknowledged
SHB.6. Missing Decrement of _numberOfStakers on Last Position Close	LOW	Fixed

SHB.7. Missing Return Value Check in ERC20 transfer and transferFrom	LOW	Fixed
SHB.8. Use of send for Transferring Native Tokens	LOW	Fixed
SHB.9. Missing Input Checks in _setToken and _setAPR Functions	LOW	Fixed
SHB.10. Use of Floating Pragma Statement	LOW	Acknowledged

3 Finding Details

SHB.1 Missing Storage Gaps in StakingUpgradeable Contract

- Severity: CRITICAL
- Status: Fixed
- Likelihood: 3
- Impact: 3

Description:

The Staking_V1 contract inherits from two abstract contracts: StakingUpgradeable and Escapable. The StakingUpgradeable contract lacks storage gaps, which makes the main contract not upgrade-safe. If any variables are added to StakingUpgradeable in a future upgrade, there's a risk of storage collision with variables from the Escapable contract.

The owner might add new state variables to the StakingUpgradeable contract in a future upgrade. This could lead to unintended overwrites of the Escapable contract's storage, potentially denial of service causing loss of funds or other unexpected behaviors.

Files Affected:

SHB.1.1: Staking_V1.sol

```
contract Staking_V1 is StakingUpgradeable, Escapable,
    ↪ AccessControlUpgradeable {
```

SHB.1.2: StakingUpgradeable.sol

```
address private _tokenAddress;
uint256 private APR;

uint256 private PRECISION;
uint256 private SECONDS_IN_YEAR;

uint256 private YEAR_1_LIMIT;
uint256 private YEAR_2_LIMIT;
```

```
uint256 private YEARLY_LIMIT;

Counters.Counter private _numberOfStakers;

uint256 public totalStaked;
uint256 public START_DATE;
uint256 public END_DATE;

mapping(address => StakingPosition[]) public stakingPositions;
mapping(address => uint256) public userStakingTotals;
mapping(address => uint256) public userPositionCount;
```

Recommendation:

Introduce storage gaps in the StakingUpgradeable contract to ensure safe upgrades in the future. This will prevent potential storage collisions and maintain the integrity of the contract's state. This can be achieved by declaring a private static array after the StakingUpgradeable variables to account for future upgrades.

Updates

The Crescite team resolved the issue by adding storage gaps in the form of a private array called __gap of size 50 to prevent storage collisions.

SHB.1.3: StakingUpgradeable.sol

```
uint256[50] private __gap;
```

SHB.2 Insufficient Funds for Staking Rewards

- Severity : HIGH
- Status : Mitigated
- Likelihood : 2
- Impact : 3

Description:

The contract does not have mechanisms in place to guarantee that it holds sufficient funds to reward stakers. This means that stakers might not receive their expected rewards if the contract's balance is insufficient.

If the contract becomes popular and attracts a large number of stakers, and the rewards are not adequately funded, early stakers might drain the contract of its funds. Later stakers, despite having staked their tokens, might find that there are no rewards left for them, leading to potential loss and dissatisfaction.

Files Affected:

SHB.2.1: StakingUpgradeable.sol

```
function calculatePositionRewards(
    uint256 positionAmount,
    uint256 positionTimestamp
) internal view returns (uint256) {
    // calculate the rewards for a year from the position amount
    uint256 rewardsPerYear = wmul(positionAmount, wdiv(wmul(APR,
        ↪ PRECISION), wmul(100, PRECISION)));

    // then calculate the rewards per second for this position
    uint256 rewardsPerSecond = wdiv(rewardsPerYear, wmul(SECONDS_IN_YEAR
        ↪ , PRECISION));

    // Calculate the time elapsed since the position was opened
    uint256 elapsedSeconds = sub(getCurrentOrEndTime(),
        ↪ positionTimestamp);

    // Calculate the rewards based on the elapsed time and rewards per
        ↪ second
    uint256 reward = wmul(rewardsPerSecond, wmul(elapsedSeconds,
        ↪ PRECISION));

    // Return the calculated rewards
```

```
    return reward;
}
```

Recommendation:

Implement mechanisms to ensure that the contract always has enough funds to reward its stakers. This could be achieved by setting aside a dedicated reward pool, periodic top-ups, or integrating checks that prevent staking if the reward pool is below a certain threshold.

Updates

The Crescite team mitigated the risk by adding a modifier that prevents opening new staking positions when the contract does not have any rewards. Additionally, the team stated that they will be funding the contract with 13.2 billion tokens to cover the maximum amount of rewards distributed over the contract lifetime given the annual staking limits.

SHB.2.2: StakingUpgradeable.sol

```
modifier rewardsPoolNotEmpty() {
  uint256 contractBalance = _token.balanceOf(address(this));
  require(contractBalance > 0, "Rewards pool is empty");
  _;
}
```

SHB.2.3: StakingUpgradeable.sol

```
function stakeTokens(
  uint256 amount
)
  external
  nonReentrant
  whenNotPaused
  nonZeroAmount(amount)
  userBalanceGte(amount)
  limitNotReached(amount)
  onlyProxy
  rewardsPoolNotEmpty
```

```
{
```

SHB.3 Precision Loss in Reward Calculation

- Severity : MEDIUM
- Status : Fixed
- Likelihood : 2
- Impact : 2

Description:

The calculatePositionRewards function in the contract calculates rewards based on the staked position and the time elapsed. However, the current calculation method introduces precision loss. The function first calculates rewardsPerYear, then divides this to get rewardsPerSecond, and finally multiplies by the elapsed seconds. This sequence of operations, especially the division followed by multiplication, leads to a loss of precision.

Stakers with smaller amounts might end up receiving zero rewards due to the precision loss. Over time, and with many stakers, this could lead to a significant amount of rewards not being distributed as intended.

Files Affected:

SHB.3.1: StakingUpgradeable.sol

```
function calculatePositionRewards(
    uint256 positionAmount,
    uint256 positionTimestamp
) internal view returns (uint256) {
    // calculate the rewards for a year from the position amount
    uint256 rewardsPerYear = wmul(positionAmount, wdiv(wmul(APR,
        ↪ PRECISION), wmul(100, PRECISION)));

    // then calculate the rewards per second for this position
    uint256 rewardsPerSecond = wdiv(rewardsPerYear, wmul(SECONDS_IN_YEAR
        ↪ , PRECISION));
```

```
    // Calculate the time elapsed since the position was opened
    uint256 elapsedSeconds = sub(getCurrentOrEndTime(), positionTimestamp);

    // Calculate the rewards based on the elapsed time and rewards per second
    uint256 reward = wmul(rewardsPerSecond, wmul(elapsedSeconds, PRECISION));

    // Return the calculated rewards
    return reward;
}
```

Recommendation:

To mitigate the precision loss, adjust the calculation sequence. Instead of dividing to get rewardsPerSecond and then multiplying, perform the multiplication first and then divide. This will ensure that the precision is maintained throughout the calculation, ensuring fair reward distribution to all stakers.

Updates

The Crescite team resolved the issue by adjusting the calculation sequence and performing multiplications before divisions.

SHB.3.2: StakingUpgradeable.sol

```
  function calculatePositionRewards(
    uint256 positionAmount,
    uint256 positionTimestamp
  ) internal view returns (uint256) {
    // Calculate the time elapsed since the position was opened
    uint256 elapsedSeconds = sub(getCurrentOrEndTime(), positionTimestamp);

```

```
    uint256 numerator = positionAmount * APR * elapsedSeconds;
    uint256 divisor = 100 * SECONDS_IN_YEAR * PRECISION;

    require(divisor > 0, "Cannot divide by zero");

    uint256 rewards = wdiv(numerator, divisor);

    // Return the calculated rewards
    return rewards;
  }
```

SHB.4 Potential DoS Due to Unoptimized Position Removal

- Severity : MEDIUM
- Status : Mitigated
- Likelihood : 1
- Impact : 3

Description:

The removeStakingPosition function in the contract is designed to remove a staking position based on its index. The current implementation shifts all elements to the left after the specified index, which can be highly inefficient and gas-intensive, especially when dealing with a large number of positions. This approach, combined with the fact that these operations are performed directly in storage using the expensive SSTORE and SLOAD opcodes, can lead to prohibitively high gas costs.

A staker with a large number of positions might attempt to close an early position (e.g., at index 0). The function would then loop over the entire array, shifting each element one step to the left. If the positions array length is substantial, the gas cost for this operation could exceed the block gas limit, effectively preventing the staker from ever closing some positions and resulting in a Denial of Service.

Files Affected:

SHB.4.1: StakingUpgradeable.sol

```
function removeStakingPosition(
    uint256 index,
    StakingPosition[] storage positions
) private returns (StakingPosition[] storage) {
    require(index < positions.length, "Index out of bounds");

    // shift elements to the left (this will delete the item at index)
    for (uint i = index; i < positions.length - 1; i++) {
      positions[i] = positions[i + 1];
    }

    // then remove the last entry
    positions.pop();

    return positions;
}
```

Recommendation:

To optimize the position removal process:

1. Instead of shifting all elements, simply swap the element to be removed with the last element in the array and then pop the last element. This ensures a constant computation time regardless of the array's length.
2. Consider working with the array in memory to reduce the gas costs associated with storage operations. After making the necessary modifications in memory, the updated array can then be written back to storage.

Updates

The Crescite team mitigated the risk, by swapping the element to be removed with the last element in the array and then popping the last element. However, the removeStakingPosition function still uses a storage reference to the positions instead of memory then reassigns the output to storage once again, which results in a gas overhead to the transaction cost.

SHB.5 Excessive Power to Owner and _escapeHatchCaller

- Severity : MEDIUM
- Status : Acknowledged
- Likelihood : 1
- Impact : 3

Description:

The contract grants significant control to two entities: the owner and the _escapeHatchCaller. The owner has the ability to withdraw all staked and reward tokens from the contract using the withdrawFunds function. Similarly, the _escapeHatchCaller can withdraw the balance of _baseTokenAddress, which could be either ERC20 tokens or native tokens, using the escapeHatch function. Such centralized control poses risks to the funds staked by users and can undermine trust in the contract.

The owner or _escapeHatchCaller could potentially drain the contract of its funds, either due to malicious intent or a compromised account. This would result in loss of funds for all stakers.

Files Affected:

SHB.5.1: StakingUpgradeable.sol

```
function withdrawFunds() external nonReentrant onlyOwner whenPaused {
    uint256 amount = IERC20(_tokenAddress).balanceOf(address(this));

    IERC20(_tokenAddress).transfer(owner(), amount);
    emit WithdrawFunds(owner(), amount);
}
```

SHB.5.2: StakingUpgradeable.sol

```
function escapeHatch() public onlyEscapeHatchCaller {
```

```
    _beforeEscapeHatch(msg.sender);

    uint total = getBalance();

    // Send the total balance of this contract to the `escapeHatchDestination`
    transfer(_escapeHatchDestination, total);

    emit EscapeHatchCalled(total);
}
```

Recommendation:

To mitigate the risks associated with centralized control:

1. Implement multi-signature requirements for critical functions like withdrawFunds and escapeHatch. This ensures that multiple trusted parties must approve any significant fund movements.
2. Consider introducing time locks or delays for these functions, giving users a window to react if they observe suspicious activity.
3. Clearly communicate the roles and responsibilities of the owner and _escapeHatchCaller to users, ensuring transparency and building trust.

Updates

The Crescite team acknowledged the issue, stating that the roles will be assigned to a multisig wallet at deployment.

SHB.6 Missing Decrement of _numberOfStakers on Last Position Close

- Severity: LOW
- Status: Fixed
- Likelihood: 2
- Impact: 1

Description:

The positionClose function allows a staker to close a staking position and claim rewards. While the function correctly updates various counters and totals, it lacks the logic to decrement the _numberOfStakers when a staker closes their last existing position. This oversight can lead to an inaccurate count of active stakers in the contract.

If multiple stakers close all their positions but the _numberOfStakers is not decremented accordingly, the contract will report a higher number of active stakers than there actually are. This can mislead other users or external systems relying on this data, potentially affecting decision-making processes based on the number of active stakers.

Files Affected:

SHB.6.1: StakingUpgradeable.sol

```
// delete the position
stakingPositions[user] = removeStakingPosition(index, stakingPositions[
    ↪ user]);

// decrement the number of positions held by user
userPositionCount[user] = sub(userPositionCount[user], 1);

// update user staked total
userStakingTotals[user] = sub(userStakingTotals[user], position.amount);

// update global staked total
totalStaked = sub(totalStaked, position.amount);
```

Recommendation:

Introduce a check in the positionClose function to determine if the staker is closing their last position. If they are, decrement the _numberOfStakers counter. This ensures that the counter accurately reflects the number of active stakers at all times.

Updates

The Crescite team resolved the issue by adding the missing check.

SHB.6.2: StakingUpgradeable.sol

```
if (userPositionCount[user] == 0) {
  _numberOfStakers.decrement();
}
```

SHB.7 Missing Return Value Check in ERC20 transfer and transferFrom

- Severity : LOW
- Status : Fixed
- Likelihood : 1
- Impact : 2

Description:

The contract uses the transfer and transferFrom methods of the ERC20 standard without checking their return values. According to the ERC20 standard, these methods should return a boolean value indicating success or failure. Ignoring these return values can lead to undetected failures in token transfers.

Exploit Scenario:

If a token transfer fails but the contract continues its execution without detecting the failure, it can lead to unintended consequences. For instance, a user might not receive their expected tokens, yet the contract behaves as if the transfer was successful. This can result in discrepancies in balances and potential loss of funds.

Files Affected:

SHB.7.1: StakingUpgradeable.sol

```
164 IERC20(_tokenAddress).transferFrom(user, address(this), amount);
```

SHB.7.2: StakingUpgradeable.sol

```
219 IERC20(_tokenAddress).transfer(user, add(position.amount, rewards));
```

SHB.7.3: StakingUpgradeable.sol

```
264 IERC20(_tokenAddress).transfer(user, add(amountToUnstake, rewards));
```

SHB.7.4: StakingUpgradeable.sol

```
307 IERC20(_tokenAddress).transfer(user, amountToTransfer);
```

SHB.7.5: StakingUpgradeable.sol

```
332 IERC20(_tokenAddress).transfer(user, rewards);
```

SHB.7.6: StakingUpgradeable.sol

```
353 IERC20(_tokenAddress).transfer(owner(), amount);
```

Recommendation:

To ensure safe and consistent interactions with ERC20 tokens, implement the use of SafeERC20 library, which provides wrappers around the standard ERC20 functions and automatically checks their return values. This will ensure that any failed token transfers are immediately detected and handled appropriately.

Updates

The Crescite team resolved the issue by implementing the use of the SafeERC20Upgradeable library.

SHB.8 Use of send for Transferring Native Tokens

- Severity : LOW
- Status : Fixed
- Likelihood : 1
- Impact : 2

Description:

The contract uses the send method to transfer native tokens. The send method forwards only 2,300 gas to the receiving contract, which might not be sufficient for more complex operations. If there are any changes in Ethereum's opcodes or gas costs in the future, this limited gas amount can lead to unexpected failures in the transfer.

Files Affected:

SHB.8.1: Escapable.sol

```
if (!payable(to).send(amount)) {
    revert('Escapable: Send failed');
}
```

Recommendation:

Replace the use of send with the call method for transferring ether, as call forwards all available gas by default. This ensures that the receiving contract has sufficient gas to execute its operations. Additionally, always check the return value of the transfer to handle any potential failures.

Updates

The Crescite team resolved the issue by performing native token transfers using call instead of send.

SHB.8.2: Escapable.sol

```
// send ETH
```

```
(bool success, ) = payable(to).call{value: amount}("");

if (!success) {
    revert("Escapable: Send failed");
}
```

SHB.9 Missing Input Checks in _setToken and _setAPR Functions

- Severity : LOW
- Status : Fixed
- Likelihood : 1
- Impact : 2

Description:

The _setToken and _setAPR functions allow the owner to set the token address and the Annual Percentage Rate (APR) respectively. However, these functions lack input validation checks. Without proper checks, there's a risk of setting invalid or unintended values.

Files Affected:

SHB.9.1: StakingUpgradeable.sol

```
function _setToken(address tokenAddress) internal onlyOwner {
    _tokenAddress = tokenAddress;

}
```

SHB.9.2: StakingUpgradeable.sol

```
function _setAPR(uint apr) internal onlyOwner {
    APR = apr;
}
```

Recommendation:

Introduce input validation checks in both functions:

1. For _setToken, ensure that the provided address is not the zero address.
2. For _setAPR, consider setting upper and lower bounds to prevent extreme values.

By implementing these checks, the contract can prevent potential errors and maintain its integrity.

Updates

The Crescite team resolved the issue by implementing a safety check on the _setToken and _setAPR arguments.

SHB.9.3: StakingUpgradeable.sol

```
function _setToken(address tokenAddress) internal onlyOwner {
    require(tokenAddress != address(0x0), "Token address cannot be the zero address");

    _token = IERC20Upgradeable(tokenAddress);
}
```

SHB.9.4: StakingUpgradeable.sol

```
function _setAPR(uint apr) internal onlyOwner {
    // Check if there are any staking positions
    require(_numberOfStakers.current() == 0, "Cannot change APR when staking positions exist");

    require(apr > 0, 'APR cannot be zero');
    require(apr <= 500, 'APR too high');

    APR = apr;
}
```

SHB.10 Use of Floating Pragma Statement

- Severity : LOW
- Status : Acknowledged
- Likelihood : 1
- Impact : 1

Description:

The contract uses a floating pragma statement, which indicates that it can be compiled with any Solidity compiler version from 0.8.17 (inclusive) up to, but not including, version 0.9.0. While this provides flexibility, it can also introduce risks if the contract is compiled with a newer compiler version that contains breaking changes or unexpected behaviors.

Files Affected:

SHB.10.1: StakingUpgradeable.sol

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.17;
```

SHB.10.2: Escapable.sol

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.17;
```

SHB.10.3: Staking_V1.sol

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.17;
```

Recommendation:

Specify a fixed compiler version in the pragma statement to ensure consistent behavior and avoid potential pitfalls introduced by newer compiler versions. For instance, if the contract was tested and audited using Solidity version 0.8.17 for example, then use pragma solidity 0.8.17; to lock in that specific version.

Updates

The Crescite team acknowledged the issue.

4 Best Practices

BP.1 Remove Unnecessary Initializations

Description:

The contract explicitly initializes the variables totalStaked and rewards with a default value of 0. In Solidity, state variables are automatically initialized to their default values. For uint256, this default is 0. This explicit setting is redundant and can make the code longer without adding any functional benefit. It's recommended to rely on Solidity's default initialization to make the code cleaner and more concise.

Files Affected:

BP.1.1: StakingUpgradeable.sol

```
87 totalStaked = 0;
```

BP.1.2: StakingUpgradeable.sol

```
433 uint256 rewards = 0;
```

Status - Fixed

BP.2 Optimize Loops for Efficiency

Description:

The loop iterating over stakingPositions arrays can be optimized for better efficiency. Caching their length into memory reduces the gas cost associated with repeatedly accessing a state variable. Additionally, using pre-increments inside an unchecked block can further reduce gas costs by avoiding overflow checks. Lastly, the loop variable i can be declared without an explicit initialization to 0, as it's the default value for uint256. It's recommended to implement these optimizations to enhance the contract's efficiency and reduce gas consumption.

Files Affected:

BP.2.1: StakingUpgradeable.sol

```
function resetUserPositionTimestamps(address user) internal {
    for (uint256 i = 0; i < stakingPositions[user].length; i++) {
      stakingPositions[user][i].timestamp = block.timestamp;
    }
}
```

BP.2.2: StakingUpgradeable.sol

```
for (uint256 i = 0; i < stakingPositions[user].length; i++) {
    uint256 amount = stakingPositions[user][i].amount;
    uint256 timestamp = stakingPositions[user][i].timestamp;

    rewards = add(rewards, calculatePositionRewards(amount, timestamp));
}
```

BP.2.3: StakingUpgradeable.sol

```
for (uint i = index; i < positions.length - 1; i++) {
  positions[i] = positions[i + 1];
}
```

Status - Partially Fixed

5 Conclusion

In this audit, we examined the design and implementation of Crescite Staking and Escapable Contracts contract and discovered several issues of varying severity. Crescite team addressed 6 issues raised in the initial report and implemented the necessary fixes, while classifying the rest as a risk with low-probability of occurrence. Shellboxes' auditors advised Crescite Team to maintain a high level of vigilance and to keep those findings in mind in order to avoid any future complications.

6 Scope Files

6.1 Audit

Files	MD5 Hash
contracts/Escapable.sol	eb185aae9753894af9a3a60c9b2a3e8b
contracts/Staking_V1.sol	7c634e11bdadcbe413c3c802e181bb36
contracts/staking/StakingUpgradeable.sol	82666aa2a9c703bdfd808ef5401b562d

6.2 Re-Audit

Files	MD5 Hash
contracts/Escapable.sol	90d7e31e362dc88678b3f6c229241a37
contracts/Staking_V1.sol	7c634e11bdadcbe413c3c802e181bb36
contracts/staking/StakingUpgradeable.sol	34af1ee4883dd5e5e1e28a4c04f9e1d3

7 Disclaimer

Shellboxes reports should not be construed as ”endorsements” or ”disapprovals” of particular teams or projects. These reports do not reflect the economics or value of any ”product” or ”asset” produced by any team or project that engages Shellboxes to do a security evaluation, nor should they be regarded as such. Shellboxes Reports do not provide any warranty or guarantee regarding the absolute bug-free nature of the examined technology, nor do they provide any indication of the technology's proprietors, business model, business or legal compliance. Shellboxes Reports should not be used in any way to decide whether to invest in or take part in a certain project. These reports don't offer any kind of investing advice and shouldn't be used that way. Shellboxes Reports are the result of a thorough auditing process designed to assist our clients in improving the quality of their code while lowering the significant risk posed by blockchain technology. According to Shellboxes, each business and person is in charge of their own due diligence and ongoing security. Shellboxes does not guarantee the security or functionality of the technology we agree to research; instead, our purpose is to assist in limiting the attack vectors and the high degree of variation associated with using new and evolving technologies.

SHELLBOXES

For a Contract Audit, contact us at contact@shellboxes.com