Tokens, Farm & Shop

Smart Contract Audit Report
Prepared for SpeedStar

Date Issued:	Apr 29, 2022
Project ID:	AUDIT2022010
Version:	v1.0
Confidentiality Level:	Public

CYBERSECURITY PROFESSIONAL SERVICE

Report Information

Project ID	AUDIT2022010
Version	v1.0
Client	SpeedStar
Project	Tokens, Farm & Shop
Auditor(s)	Natsasit Jirathammanuwat Puttimet Thammasaeng
Author(s)	Natsasit Jirathammanuwat
Reviewer	Pongsakorn Sommalai
Confidentiality Level	Public

Version History

Version	Date	Description	Author(s)
1.0	Apr 29, 2022	Full report	Natsasit Jirathammanuwat

Contact Information

Company	Inspex
Phone	(+66) 90 888 7186
Telegram	t.me/inspexco
Email	audit@inspex.co

Table of Contents

1. Executive Summary

As requested by SpeedStar, Inspex team conducted an audit to verify the security posture of the Tokens, Farm & Shop smart contracts between Feb 9, 2022 and Feb 11, 2022. During the audit, Inspex team examined all smart contracts and the overall operation within the scope to understand the overview of Tokens, Farm & Shop smart contracts. Static code analysis, dynamic analysis, and manual review were done in conjunction to identify smart contract vulnerabilities together with technical & business logic flaws that may be exposed to the potential risk of the platform and the ecosystem. Practical recommendations are provided according to each vulnerability found and should be followed to remediate the issue.

1.1. Audit Result

In the initial audit, Inspex found 2 critical, 10 high, 4 medium, 1 low, 1 very low, and 4 info-severity issues. With the project team's prompt response in resolving the issues found by Inspex, all issues were resolved or mitigated in the reassessment. Therefore, Inspex trusts that Tokens, Farm & Shop smart contracts have high-level protections in place to be safe from most attacks.

1.2. Disclaimer

This security audit is not produced to supplant any other type of assessment and does not guarantee the discovery of all security vulnerabilities within the scope of the assessment. However, we warrant that this audit is conducted with goodwill, professional approach, and competence. Since an assessment from one single party cannot be confirmed to cover all possible issues within the smart contract(s), Inspex suggests conducting multiple independent assessments to minimize the risks. Lastly, nothing contained in this audit report should be considered as investment advice.

2. Project Overview

2.1. Project Introduction

SpeedStar is a simulation game in which users can take on the role of a player in the Starverse universe. In this universe, users can be anything they want. The game is developed by the Hell Factory team and launched on the Harmony One Chain.

SpeedStar Farm allows users to stake their NFT i.e., horse, facility, and stable in order to earn the $SPEED. The users can purchase NFT horses and facilities to use in the SpeedStar platform via the Shop contract.

Scope Information:

Project Name	Tokens, Farm & Shop
Website	https://speedstargame.com/
Smart Contract Type	Ethereum Smart Contract
Chain	Harmony One Chain
Programming Language	Solidity

Audit Information:

Audit Method	Whitebox
Audit Date	Feb 9, 2022 - Feb 11, 2022
Reassessment Date	Feb 24, 2022

The audit method can be categorized into two types depending on the assessment targets provided:

1. **Whitebox**: The complete source code of the smart contracts are provided for the assessment.
2. **Blackbox**: Only the bytecodes of the smart contracts are provided for the assessment.

2.2. Scope

The following smart contracts were audited and reassessed by Inspex in detail:

Initial Audit: (Commit: 9d2450297515f302fca000275d4c1a47afcf909d)

Contract	Location (URL)
Staking	https://github.com/HellFactory/speedstar-audit/blob/9d24502975/contracts/farm/Staking.sol
JOC	https://github.com/HellFactory/speedstar-audit/blob/9d24502975/contracts/tokens/JOC.sol
Speed	https://github.com/HellFactory/speedstar-audit/blob/9d24502975/contracts/tokens/Speed.sol
Star	https://github.com/HellFactory/speedstar-audit/blob/9d24502975/contracts/tokens/Star.sol
Facility	https://github.com/HellFactory/speedstar-audit/blob/9d24502975/contracts/Facility.sol
Horse	https://github.com/HellFactory/speedstar-audit/blob/9d24502975/contracts/Horse.sol
Shop	https://github.com/HellFactory/speedstar-audit/blob/9d24502975/contracts/Shop.sol

Reassessment: (Commit: 3e39d7acf9c1aa9f3a5511c161c2035ba7d6bc1f)

Contract	Location (URL)
Staking	https://github.com/HellFactory/-speedstar-audit/blob/3e39d7acf9/contracts/farm/Staking.sol
JOC	https://github.com/HellFactory/-speedstar-audit/blob/3e39d7acf9/contracts/tokens/JOC.sol
Speed	https://github.com/HellFactory/-speedstar-audit/blob/3e39d7acf9/contracts/tokens/Speed.sol
Star	https://github.com/HellFactory/-speedstar-audit/blob/3e39d7acf9/contracts/tokens/Star.sol
Facility	https://github.com/HellFactory/-speedstar-audit/blob/3e39d7acf9/contracts/Facility.sol
Horse	https://github.com/HellFactory/-speedstar-audit/blob/3e39d7acf9/contracts/Horse.sol

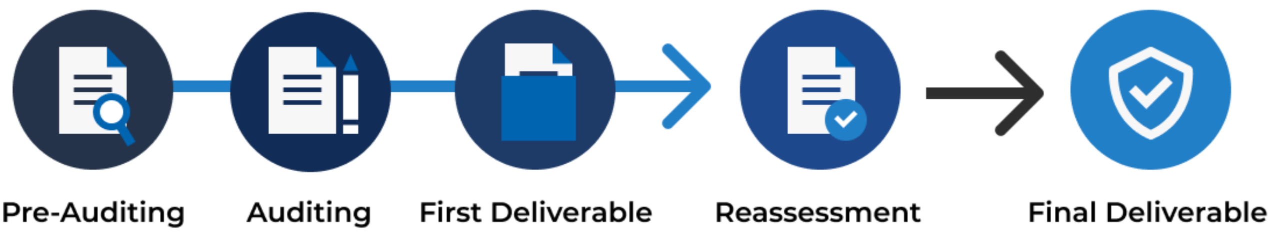
Shop	https://github.com/HellFactory/-speedstar-audit/blob/3e39d7acf9/contracts/Shop.sol

The assessment scope covers only the in-scope smart contracts and the smart contracts that they inherit from.

3. Methodology

Inspex conducts the following procedure to enhance the security level of our clients' smart contracts:

1. **Pre-Auditing**: Getting to understand the overall operations of the related smart contracts, checking for readiness, and preparing for the auditing
2. **Auditing**: Inspecting the smart contracts using automated analysis tools and manual analysis by a team of professionals
3. **First Deliverable and Consulting**: Delivering a preliminary report on the findings with suggestions on how to remediate those issues and providing consultation
4. **Reassessment**: Verifying the status of the issues and whether there are any other complications in the fixes applied
5. **Final Deliverable**: Providing a full report with the detailed status of each issue



3.1. Test Categories

Inspex smart contract auditing methodology consists of both automated testing with scanning tools and manual testing by experienced testers. We have categorized the tests into 3 categories as follows:

1. **General Smart Contract Vulnerability (General)** - Smart contracts are analyzed automatically using static code analysis tools for general smart contract coding bugs, which are then verified manually to remove all false positives generated.
2. **Advanced Smart Contract Vulnerability (Advanced)** - The workflow, logic, and the actual behavior of the smart contracts are manually analyzed in-depth to determine any flaws that can cause technical or business damage to the smart contracts or the users of the smart contracts.
3. **Smart Contract Best Practice (Best Practice)** - The code of smart contracts is then analyzed from the development perspective, providing suggestions to improve the overall code quality using standardized best practices.

3.2. Audit Items

The following audit items were checked during the auditing activity.

General
Reentrancy Attack
Integer Overflows and Underflows
Unchecked Return Values for Low-Level Calls
Bad Randomness
Transaction Ordering Dependence
Time Manipulation
Short Address Attack
Outdated Compiler Version
Use of Known Vulnerable Component
Deprecated Solidity Features
Use of Deprecated Component
Loop with High Gas Consumption
Unauthorized Self-destruct
Redundant Fallback Function
Insufficient Logging for Privileged Functions
Invoking of Unreliable Smart Contract
Use of Upgradable Contract Design
Centralized Control of State Variable
Advanced
Business Logic Flaw
Ownership Takeover
Broken Access Control
Broken Authentication

Improper Kill-Switch Mechanism
Improper Front-end Integration
Insecure Smart Contract Initiation
Denial of Service
Improper Oracle Usage
Memory Corruption
Best Practice
Use of Variadic Byte Array
Implicit Compiler Version
Implicit Visibility Level
Implicit Type Inference
Function Declaration Inconsistency
Token API Violation
Best Practices Violation

3.3. Risk Rating

OWASP Risk Rating Methodology[1] is used to determine the severity of each issue with the following criteria:

- **Likelihood**: a measure of how likely this vulnerability is to be uncovered and exploited by an attacker.
- **Impact**: a measure of the damage caused by a successful attack

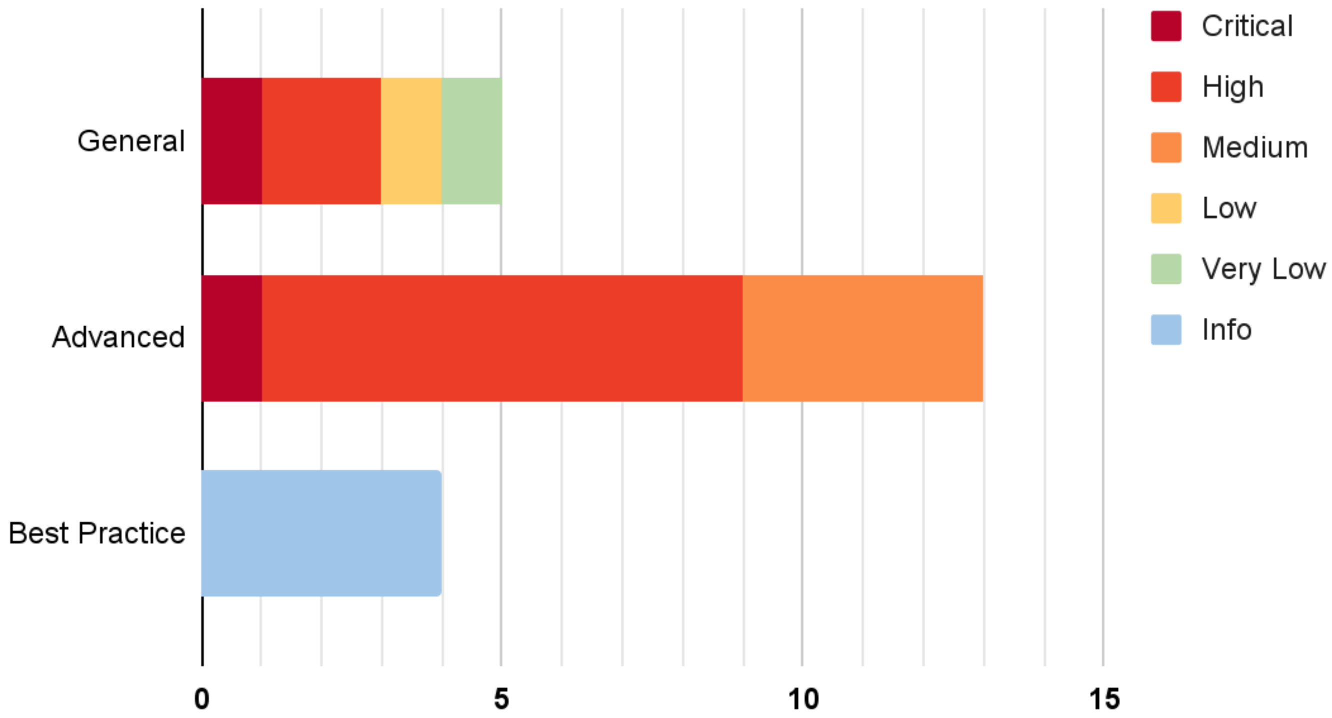
Both likelihood and impact can be categorized into three levels: **Low**, **Medium**, and **High**.

Severity is the overall risk of the issue. It can be categorized into five levels: **Very Low**, **Low**, **Medium**, **High**, and **Critical**. It is calculated from the combination of likelihood and impact factors using the matrix below. The severity of findings with no likelihood or impact would be categorized as **Info**.

Likelihood Impact	Low	Medium	High
Low	Very Low	Low	Medium
Medium	Low	Medium	High
High	Medium	High	Critical

4. Summary of Findings

From the assessments, Inspex has found 22 issues in three categories. The following chart shows the number of the issues categorized into three categories: **General**, **Advanced**, and **Best Practice**.



The statuses of the issues are defined as follows:

Status	Description
Resolved	The issue has been resolved and has no further complications.
Resolved *	The issue has been resolved with mitigations and clarifications. For the clarification or mitigation detail, please refer to Chapter 5.
Acknowledged	The issue's risk has been acknowledged and accepted.
No Security Impact	The best practice recommendation has been acknowledged.

The information and status of each issue can be found in the following table:

ID	Title	Category	Severity	Status
IDX-001	Reentrancy Attack	General	Critical	Resolved
IDX-002	Broken Access Control in withdrawHorseInStable() Function	Advanced	Critical	Resolved
IDX-003	Manual Minting by Privileged Role	General	High	Resolved *
IDX-004	Missing user.rewardDebt State Update After payReward()	Advanced	High	Resolved
IDX-005	Double Reward Payout in withdrawStable() Function	Advanced	High	Resolved
IDX-006	Missing pool.totalStake State Update	Advanced	High	Resolved
IDX-007	Improper Horse.bornAt Value Assignment	Advanced	High	Resolved
IDX-008	Miscalculation in calculateRewardAndUpdateRemainHorse() Function	Advanced	High	Resolved
IDX-009	Improper runningBlock Value Calculation	Advanced	High	Resolved
IDX-010	Centralized Control of State Variable	General	High	Resolved *
IDX-011	Improper claim() Function Implementation	Advanced	High	Resolved
IDX-012	Missing stable.multiplier Multiplication in Reward Calculation	Advanced	High	Resolved
IDX-013	Missing Native Token Withdrawal Function	Advanced	Medium	Resolved
IDX-014	Improper horseLimitStaking() Function Implementation	Advanced	Medium	Resolved
IDX-015	Incorrect Price Incremental Calculation in buyPack() Function	Advanced	Medium	Resolved
IDX-016	Improper Sale Properties Modification During On-Going Sale Event	Advanced	Medium	Resolved
IDX-017	Loop Over Unbounded Data Structure	General	Low	Resolved
IDX-018	Insufficient Logging for Privileged Functions	General	Very Low	Resolved
IDX-019	Inexplicit Solidity Compiler Version	Best Practice	Info	Resolved

IDX-020	Improper Function Visibility	Best Practice	**Info**	Resolved
IDX-021	Incorrect Logging Parameter	Best Practice	**Info**	Resolved
IDX-022	Use of transfer() Function to Transfer Native Token	Best Practice	**Info**	Resolved

* The mitigations or clarifications by SpeedStar can be found in Chapter 5.

5. Detailed Findings Information

5.1. Reentrancy Attack

ID	IDX-001
Target	Staking
Category	General Smart Contract Vulnerability
CWE	CWE-841: Improper Enforcement of Behavioral Workflow
Risk	**Severity: Critical** **Impact: High** The reward token can be claimed multiple times resulting in reward draining from the `Staking` contract. When there is no reward in the contract, this also results in a denial of service on all deposit and withdraw functions. **Likelihood: High** It is very likely that the attacker will perform this attack by staking at least a facility or a horse then withdrawing it from the `Staking` contract.
Status	Resolved SpeedStar team has resolved this issue as suggested by using the `nonReentrant` modifier from the `ReentrancyGuard` contract[3] of the OpenZeppalin and implementing the check-effects-interactions pattern in commit `3e39d7acf9c1aa9f3a5511c161c2035ba7d6bc1f`.

5.1.1. Description

The `ERC721.safeTransferFrom()` functions are called to withdraw the NFT from the `Staking` contract to the user. Then `onERC721Received()` callback function will be called when the NFT was transferred. The attacker can create a malicious contract which implement `onERC721Received()` function to perform the reentrancy attack and drain all the reward from the contract as the following scenario:

1. Create a contract and transfer a stable and a horse to the contract.
2. Contract calls the `depositStable()` function to stake a stable.
3. Contract calls the `depositHorseInStable()` function to stake a horse in the stable.
4. After the reward was updated, the contract calls the `withdrawHorseInStable()` function.
5. The reward is then transferred to the contract via the `payReward()` function and the horse is transferred to the contract via the `safeTransferFrom()` function.
6. The `onERC721Received()` callback function will be triggered and the contract will call the `depositHorseInStable()` function again the reward will payout without `user.rewardDebt` updated.
7. Perform step 3-6 until all reward is drained from the contract.

The following table shows all effected functions:

Target	Contract	Function
Staking.sol(L:521)	Staking	withdrawFacility()
Staking.sol(L:585)	Staking	withdrawStable()
Staking.sol(L:665)	Staking	withdrawHorseInStable()
Staking.sol(L:748)	Staking	withdrawHorse()

5.1.2. Remediation

Implementing the check-effects-interactions pattern or using the `nonReentrant` modifier from the `ReentrancyGuard` contract[3] of the OpenZeppalin.

5.2. Broken Access Control in withdrawHorseInStable() Function

ID	IDX-002
Target	Staking
Category	Advanced Smart Contract Vulnerability
CWE	CWE-284: Improper Access Control
Risk	**Severity: Critical** **Impact: High** Due to improper authorization checks in the `withdrawHorseInStable()` function, any staked horses in the `Staking` contract can be withdrawn by the attacker. **Likelihood: High** It is very likely that the attacker who stakes at least one horse in a stable (already called the `depositHorseInStable()` function) can steal a horse that is staked in the contract.
Status	**Resolved** SpeedStar team has resolved this issue as suggested by checking the ownership of the horse in commit `3e39d7acf9c1aa9f3a5511c161c2035ba7d6bc1f`.

5.2.1. Description

In the `withdrawHorseInStable()` function, the ownership of the horse with `_horseTokenId` is not checked properly. The attacker can call the `withdrawHorseInStable()` function with attacker's `_stableTokenId` and victim's `_horseTokenId`. When the function is called, only the ownership of the stable is checked in line 653, and then the horse is transferred to the `msg.sender` without any ownership checks as shown in line 665.

Staking.sol

```
647 function withdrawHorseInStable(
648     uint256 _stableTokenId,
649     uint256 _horseTokenId
650 ) public {
651     PoolInfo storage pool = poolInfo;
652     UserInfo storage user = userInfo[msg.sender];
653     require(user.ownedStable[_stableTokenId], "No stable staking");
654 
655     updatePool();
656     uint256 pending = user.amount.mul(pool.accSpeedPerShare).div(1e12).sub(
657         user.rewardDebt
658     );
659 
660     if (pending > 0) {
661         payReward(user);
662     }
```

```
    Stable storage stable = user.stables[user.stableIndex[_stableTokenId]];

    horse.safeTransferFrom(
        address(this),
        address(msg.sender),
        _horseTokenId
    );
    // decrease amount from this stable
    user.amount = user.amount.sub(getPopularityInStable(_stableTokenId));
    // remove horse instable
    user.ownedTokenId[_horseTokenId] = false;
    removeHorseFromList(stable.horses, stable.horseIndex[_horseTokenId]);
    delete stable.horseIndex[_horseTokenId];
    // update amount
    user.amount = user.amount.add(getPopularityInStable(_stableTokenId));
    user.rewardDebt = user.amount.mul(pool.accSpeedPerShare).div(1e12);

    emit WithdrawHorseInStable(
        msg.sender,
        _stableTokenId,
        horse.getPopularity(_horseTokenId),
        _horseTokenId
    );
}
```

5.2.2. Remediation

Inspex suggests checking the ownership of the horse with `user.ownedTokenId[_horseTokenId]` as shown in line 654:

Staking.sol

```
647 function withdrawHorseInStable(
648     uint256 _stableTokenId,
649     uint256 _horseTokenId
650 ) public {
651     PoolInfo storage pool = poolInfo;
652     UserInfo storage user = userInfo[msg.sender];
653     require(user.ownedStable[_stableTokenId], "No stable staking");
654     require(user.ownedTokenId[_horseTokenId], "No horse staking in the stable");
```

5.3. Manual Minting by Privileged Role

ID	IDX-003
Target	JOC Speed Star Facility Horse
Category	General Smart Contract Vulnerability
CWE	CWE-284: Improper Access Control
Risk	**Severity: High** **Impact: High** The admin role can mint the tokens and NFTs without any restriction. **Likelihood: Medium** The contract owner can set any wallet address to be an admin role which can mint the tokens or the NFTs freely. It is likely for the owner to profit from this action.
Status	Resolved * SpeedStar team has confirmed that minting authority of all tokens and NFTs will be transferred to the related contract only which contains the minting logic such as game rewarding, horse breeding, or facility shop. The platform users must monitor the minter of affected contracts in order to confirm that only the trusted contracts have the minting authority before and during using the platform.

5.3.1. Description

The following table shows all manual minting functions:

Target	Function	Modifier
JOC.sol(L:906)	mint()	isAdmin
Speed.sol(L:906)	mint()	isAdmin
Star.sol(L:909)	mint()	isAdmin
Facility.sol(L:40)	mintStable()	onlyOwner
Facility.sol(L:55)	mintStables()	onlyOwner
Facility.sol(L:73)	mintFacility()	onlyOwner

Facility.sol(L:88)	mintFacilitys()	onlyOwner
Horse.sol(L:41)	mint()	onlyOwner
Horse.sol(L:57)	mints()	onlyOwner

For example, the `mint()` function of the `Speed` contract has the `isAdmin` modifier. This means that the admin of the `Speed` contract can manually mint the $SPEED anytime they want, as shown in the following source code:

Speed.sol

```
function mint(address _receiver, uint256 _amount) external isAdmin {
    _mint(_receiver, _amount);
}
```

5.3.2. Remediation

For the JOC and Star contracts:

Inspex suggests removing the admin role from the contract, setting the `onlyOwner` as the modifier of minting functions, and setting the owner of the contract as trusted contract only.

For the Speed contract:

Inspex suggests removing the admin role from the contract, setting the `onlyOwner` as the modifier of minting functions and setting the owner of the contract as `Staking` contract only.

For the Facility and the Horse contracts:

Inspex suggests implementing the provably-fair and verifiable random in the `Shop` contract and giving the minting authority to the `Shop` contract only.

Further information about verifiable random on Harmony can be found at Harmony VRF [2].

5.4. Missing user.rewardDebt State Update After payReward()

ID	IDX-004
Target	Staking
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** High **Impact:** Medium The `user.rewardDebt` state is not updated after claiming the reward in the `depositFacility()` and `depositStable()` functions. Thus, the attacker can call these functions repeatedly to drain an entire reward in the `Staking` contract. When there is no reward in the contract, this also results in a denial of service on all deposit and withdraw functions. **Likelihood:** High The `depositFacility()` and `depositStable()` functions can be executed by anyone, so there is no restriction to prevent this issue.
Status	Resolved SpeedStar team has resolved this issue as suggested by updating the `user.rewardDebt` state after the user claims the reward in commit `3e39d7acf9c1aa9f3a5511c161c2035ba7d6bc1f`.

5.4.1. Description

The `payReward()` function is called in the `depositFacility()` function as shown in line 492:

Staking.sol

```
486 function depositFacility(uint256 _tokenId) external {
487     UserInfo storage user = userInfo[msg.sender];
488     require(!user.ownedFacility[_tokenId], "Already staking");
489 
490     updatePool();
491     if (user.amount > 0) {
492         payReward(user);
493     }
494 
495     facility.safeTransferFrom(address(msg.sender), address(this), _tokenId);
496 
497     // update amount
498     uint256 popularity = facility.popularity(_tokenId);
499     user.amount = user.amount.add(popularity);
500     user.ownedFacility[_tokenId] = true;
```

```

    user.facilityIndex[_tokenId] = user.facility.length;
    user.facility.push(_tokenId);

    emit DepositFacility(msg.sender, 0, _tokenId);
}
```

The `payReward()` function is called in the `depositStable()` function as shown in line 547:

Staking.sol

```
541 function depositStable(uint256 _tokenId) external {
542     UserInfo storage user = userInfo[msg.sender];
543     require(!user.ownedStable[_tokenId], "Already staking");
544
545     updatePool();
546     if (user.amount > 0) {
547         payReward(user);
548     }
549     uint256 multiplier = facility.multipliers(_tokenId);
550     facility.safeTransferFrom(address(msg.sender), address(this), _tokenId);
551
552     Stable[] storage userStable = user.stables;
553     user.stableIndex[_tokenId] = userStable.length;
554     user.ownedStable[_tokenId] = true;
555     userStable.push();
556
557     Stable storage newStable = userStable[user.stableIndex[_tokenId]];
558     newStable.tokenId = _tokenId;
559     newStable.multiplier = multiplier;
560
561     emit DepositStable(msg.sender, 0, _tokenId);
562 }
```

The source code of two functions above shows that the `user.rewardDebt` is not updated after the user claims reward, resulting in reward drained from the contract.

5.4.2. Remediation

Inspex suggests updating the `user.rewardDebt` state after the user claims the reward. For example updating the `user.rewardDebt` state as shown in line 494 and line 549:

Staking.sol

```
486 function depositFacility(uint256 _tokenId) external {
487     UserInfo storage user = userInfo[msg.sender];
488     require(!user.ownedFacility[_tokenId], "Already staking");
489 
490     updatePool();
491     if (user.amount > 0) {
492         payReward(user);
493     }
494     user.rewardDebt = user.amount.mul(pool.accSpeedPerShare).div(1e12);
495     facility.safeTransferFrom(address(msg.sender), address(this), _tokenId);
496 
497     // update amount
498     uint256 popularity = facility.popularity(_tokenId);
499     user.amount = user.amount.add(popularity);
500     user.ownedFacility[_tokenId] = true;
501 
502     user.facilityIndex[_tokenId] = user.facility.length;
503     user.facility.push(_tokenId);
504 
505     emit DepositFacility(msg.sender, 0, _tokenId);
506 }
```

Staking.sol

```
541 function depositStable(uint256 _tokenId) external {
542     UserInfo storage user = userInfo[msg.sender];
543     require(!user.ownedStable[_tokenId], "Already staking");
544 
545     updatePool();
546     if (user.amount > 0) {
547         payReward(user);
548     }
549     user.rewardDebt = user.amount.mul(pool.accSpeedPerShare).div(1e12);
550     uint256 multiplier = facility.multipliers(_tokenId);
551     facility.safeTransferFrom(address(msg.sender), address(this), _tokenId);
552 
553     Stable[] storage userStable = user.stables;
554     user.stableIndex[_tokenId] = userStable.length;
555     user.ownedStable[_tokenId] = true;
556     userStable.push();
557 
558     Stable storage newStable = userStable[user.stableIndex[_tokenId]];
```

```
    newStable.tokenId = _tokenId;
    newStable.multiplier = multiplier;

    emit DepositStable(msg.sender, 0, _tokenId);
}
```

5.5. Double Reward Payout in withdrawStable() Function

ID	IDX-005
Target	Staking
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity: High** **Impact: Medium** The reward will be paid twice when the `withdrawStable()` function is called, resulting in reward drained from the contract. When there is no reward in the contract, this also results in a denial of service on all deposit/withdraw functions. **Likelihood: High** The issue occurs every time when the `withdrawStablewithdrawStable()` function is called by anyone who wants to withdraw their stables.
Status	**Resolved** SpeedStar team has resolved this issue as suggested by updating the `rewardDebt` state after distributing the reward in commit `3e39d7acf9c1aa9f3a5511c161c2035ba7d6bc1f`.

5.5.1. Description

Firstly, the `payReward()` function is called in the `withdrawStable()` function as shown in line 574. Then, the `withdrawHorseInStable()` function is called inside the for-loop at line 582:

Staking.sol

```
564 function withdrawStable(uint256 _stableTokenId) external {
565     PoolInfo storage pool = poolInfo;
566     UserInfo storage user = userInfo[msg.sender];
567     require(user.ownedStable[_stableTokenId], "No stable staking");
568 
569     updatePool();
570     uint256 pending = user.amount.mul(pool.accSpeedPerShare).div(1e12).sub(
571         user.rewardDebt
572     );
573     if (pending > 0) {
574         payReward(user);
575     }
576 
577     Stable[] storage userStable = user.stables;
578     Stable storage stable = userStable[user.stableIndex[_stableTokenId]];
579     // unstake all horse in stable
580     Horse[] memory horses = stable.horses;
```

```
    for (uint256 index = 0; index < horses.length; index++) {
        withdrawHorseInStable(_stableTokenId, horses[index].tokenId);
    }
    // transfer stable to user
    facility.safeTransferFrom(
        address(this),
        address(msg.sender),
        _stableTokenId
    );
    // update stable data instead.
    stable = user.stables[userStable.length - 1];
    user.stableIndex[stable.tokenId] = user.stableIndex[_stableTokenId];
    user.stables.pop();
    user.ownedStable[_stableTokenId] = false;
    user.rewardDebt = user.amount.mul(pool.accSpeedPerShare).div(1e12);

    emit WithdrawStable(msg.sender, _stableTokenId);
}
```

Before setting the `user.rewardDebt` state, the `payReward()` function is called again in the `withdrawHorseInStable()` function line 661. Then, `user.rewardDebt` will be updated in line 678.

Staking.sol

```
647 function withdrawHorseInStable(
648     uint256 _stableTokenId,
649     uint256 _horseTokenId
650 ) public {
651     PoolInfo storage pool = poolInfo;
652     UserInfo storage user = userInfo[msg.sender];
653     require(user.ownedStable[_stableTokenId], "No stable staking");
654 
655     updatePool();
656     uint256 pending = user.amount.mul(pool.accSpeedPerShare).div(1e12).sub(
657         user.rewardDebt
658     );
659 
660     if (pending > 0) {
661         payReward(user);
662     }
663     Stable storage stable = user.stables[user.stableIndex[_stableTokenId]];
664 
665     horse.safeTransferFrom(
666         address(this),
667         address(msg.sender),
668         _horseTokenId
669     );
670     // decrease amount from this stable
```

```
    user.amount = user.amount.sub(getPopularityInStable(_stableTokenId));
    // remove horse instable
    user.ownedTokenId[_horseTokenId] = false;
    removeHorseFromList(stable.horses, stable.horseIndex[_horseTokenId]);
    delete stable.horseIndex[_horseTokenId];
    // update amount
    user.amount = user.amount.add(getPopularityInStable(_stableTokenId));
    user.rewardDebt = user.amount.mul(pool.accSpeedPerShare).div(1e12);

    emit WithdrawHorseInStable(
        msg.sender,
        _stableTokenId,
        horse.getPopularity(_horseTokenId),
        _horseTokenId
    );
}
```

5.5.2. Remediation

Inspex suggests updating the `user.rewardDebt` suddenly after claiming the reward with the `payReward()` function as shown in line 576:

Staking.sol

```
564 function withdrawStable(uint256 _stableTokenId) external {
565     PoolInfo storage pool = poolInfo;
566     UserInfo storage user = userInfo[msg.sender];
567     require(user.ownedStable[_stableTokenId], "No stable staking");
568
569     updatePool();
570     uint256 pending = user.amount.mul(pool.accSpeedPerShare).div(1e12).sub(
571         user.rewardDebt
572     );
573     if (pending > 0) {
574         payReward(user);
575     }
576     user.rewardDebt = user.amount.mul(pool.accSpeedPerShare).div(1e12);
577     Stable[] storage userStable = user.stables;
578     Stable storage stable = userStable[user.stableIndex[_stableTokenId]];
579     // unstake all horse in stable
580     Horse[] memory horses = stable.horses;
581     for (uint256 index = 0; index < horses.length; index++) {
582         withdrawHorseInStable(_stableTokenId, horses[index].tokenId);
583     }
584     // transfer stable to user
585     facility.safeTransferFrom(
586         address(this),
587         address(msg.sender),
588         _stableTokenId
589     );
590     // update stable data instead.
591     stable = user.stables[userStable.length - 1];
592     user.stableIndex[stable.tokenId] = user.stableIndex[_stableTokenId];
593     user.stables.pop();
594     user.ownedStable[_stableTokenId] = false;
595
596     emit WithdrawStable(msg.sender, _stableTokenId);
597 }
```

5.6. Missing pool.totalStake State Update

ID	IDX-006
Target	Staking
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** High **Impact:** Medium The `pool.totalStake` state is not updated properly, resulting in the `pool.accSpeedPerShare` state is higher than the actual amount and the reward being distributed more than expected. When there is no reward in the contract, this also results in a denial of service on all deposit and withdraw functions. **Likelihood:** High The issue occurs every time when the `updatePool()` function is called when the users deposit or withdraw horse as an example.
Status	Resolved SpeedStar team has resolved this issue as suggested by updating the `pool.totalStake` state in commit `3e39d7acf9c1aa9f3a5511c161c2035ba7d6bc1f`.

5.6.1. Description

The `pool.totalStake` state is updated in the `depositHorse()` and `withdrawHorse()` functions only. So, when the `user.amount` is updated at line 499, the `pool.totalStake` is not updated respectively. This results in the miscalculation when claiming the rewards.

Staking.sol

```
486 function depositFacility(uint256 _tokenId) external {
487     UserInfo storage user = userInfo[msg.sender];
488     require(!user.ownedFacility[_tokenId], "Already staking");
489 
490     updatePool();
491     if (user.amount > 0) {
492         payReward(user);
493     }
494 
495     facility.safeTransferFrom(address(msg.sender), address(this), _tokenId);
496 
497     // update amount
498     uint256 popularity = facility.popularity(_tokenId);
499     user.amount = user.amount.add(popularity);
500     user.ownedFacility[_tokenId] = true;
```

```

    user.facilityIndex[_tokenId] = user.facility.length;
    user.facility.push(_tokenId);

    emit DepositFacility(msg.sender, 0, _tokenId);
}
```

The following table shows all functions with updating of the `user.amount` state without the `pool.totalStake` state updated respectively:

Target	Contract	Function
Staking.sol(L:379)	Staking	payReward()
Staking.sol(L:499)	Staking	depositFacility()
Staking.sol(L:535)	Staking	withdrawFacility()
Staking.sol(L:636)	Staking	depositHorseInStable()
Staking.sol(L:671, 677)	Staking	withdrawHorseInStable()

5.6.2. Remediation

Inspex suggests updating the `pool.totalStake` every time that the `user.amount` is updated, for example as shown in line 499 and 500:

Staking.sol

```
486 function depositFacility(uint256 _tokenId) external {
487     UserInfo storage user = userInfo[msg.sender];
488     require(!user.ownedFacility[_tokenId], "Already staking");
489 
490     updatePool();
491     if (user.amount > 0) {
492         payReward(user);
493     }
494 
495     facility.safeTransferFrom(address(msg.sender), address(this), _tokenId);
496 
497     // update amount
498     uint256 popularity = facility.popularity(_tokenId);
499     user.amount = user.amount.add(popularity);
500     pool.totalStake = pool.totalStake.add(popularity);
501     user.ownedFacility[_tokenId] = true;
502 
503     user.facilityIndex[_tokenId] = user.facility.length;
504     user.facility.push(_tokenId);
505 
506     emit DepositFacility(msg.sender, 0, _tokenId);
507 }
```

5.7. Improper Horse.bornAt Value Assignment

ID	IDX-007
Target	Horse
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity: High** **Impact: Medium** The `bornAt` state of the minted horse is updated when `mint()` function is executed. Thus, the `bornAt` state of the minted horse will be updated when the new horse was mint which leads to retired horse miscalculation. **Likelihood: High** The issue occurs every time when the `Horse` token is minted.
Status	**Resolved** SpeedStar team has resolved this issue as suggested by implementing the `bornAt` state for each horse in commit `3e39d7acf9c1aa9f3a5511c161c2035ba7d6bc1f`.

5.7.1. Description

In the `Horse` contract, the `mint()` function is used for mint the horse as shown below in line 51, the `bornAt` state is shared among every horse. Since the `bornAt` state is updated every time when the `mint()` function is executed, the `bornAt` state of the minted horse is updated too.

Horse.sol

```
41 function mint(
42     address _receiver,
43     string memory _uri,
44     uint256 _tokenId,
45     uint256 _rarity,
46     uint256 _age
47 ) public onlyOwner {
48     _mint(_receiver, _tokenId);
49     uri[_tokenId] = _uri;
50     rarity[_tokenId] = _rarity;
51     bornAt = block.number;
52     age[_tokenId] = _age;
53 
54     emit Mint(_receiver, _tokenId);
55 }
```

As a result, retired horses were miscalculated. Because the `bornAt` state is used in the `getRemainAge()` function to calculate the horse's remaining age.

5.7.2. Remediation

Inspex suggests separating the `bornAt` state for each horse by implementing the `bornAt` state for each NFT and assigning it when `mint()` function is called, for example:

Horse.sol

```
Counters.Counter private _tokenIds;
string public baseURI;
mapping(uint256 => string) private uri;
mapping(uint256 => uint256) private rarity;
mapping(uint256 => uint256) private age;
mapping(uint256 => uint256) public bornAt;
uint256 public retriedAge;
```

Horse.sol

```
function mint(
    address _receiver,
    string memory _uri,
    uint256 _tokenId,
    uint256 _rarity,
    uint256 _age
) public onlyOwner {
    _mint(_receiver, _tokenId);
    uri[_tokenId] = _uri;
    rarity[_tokenId] = _rarity;
    bornAt[_tokenId] = block.number;
    age[_tokenId] = _age;

    emit Mint(_receiver, _tokenId);
}
```

5.8. Miscalculation in calculateRewardAndUpdateRemainHorse() Function

ID	IDX-008
Target	Staking
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** High **Impact:** Medium With the current design, the popularity of retired horses will drop to 20% of the previous reward every time that `calculateRewardAndUpdateRemainHorse()` function is called. **Likelihood:** High This issue occurs whenever the user claims the reward or deposits/withdraws NFTs through the `Staking` contract.
Status	Resolved SpeedStar team has resolved this issue as suggested by updating the `_horse.popularity` state with the state from the `getPopularity()` function in the `Horse` contract in commit `3e39d7acf9c1aa9f3a5511c161c2035ba7d6bc1f`.

5.8.1. Description

In the `calculateRewardAndUpdateRemainHorse()` function, the reward for the retired horse is calculated by `rewardPerBlock` divided by 5 as shown in line 431. However the `_horse.popularity` state is also assigned with 1/5 of `_horse.popularity` again in line 443 which results in reward miscalculation.

Staking.sol

```
419 function calculateRewardAndUpdateRemainHorse(
420     Horse[] storage _horses,
421     uint256 _rewardPerAmount
422 ) internal returns (uint256, uint256) {
423     uint256 normalizeReward;
424     uint256 totalPopularlity;
425
426     for (uint256 index = 0; index < _horses.length; index++) {
427         Horse storage _horse = _horses[index];
428
429         uint256 runningBlock = block.number - _horse.enterBlock;
430         uint256 rewardPerBlock = _rewardPerAmount.div(runningBlock);
431         uint256 retriedReward = rewardPerBlock.div(5);
432
433         if (runningBlock > _horse.remainBlock) {
434             uint256 retriedBlock = runningBlock - _horse.remainBlock;
```

```
            normalizeReward = normalizeReward.add(
                _horse.remainBlock.mul(rewardPerBlock).mul(
                    _horse.popularity
                )
            );
            normalizeReward = normalizeReward.add(
                retriedBlock.mul(retriedReward).mul(_horse.popularity)
            );
            _horse.popularity = _horse.popularity.div(5);

            _horse.remainBlock = 0;
        } else {
            normalizeReward = normalizeReward.add(
                runningBlock.mul(rewardPerBlock).mul(_horse.popularity) // 2x
            );
            _horse.remainBlock = _horse.remainBlock.sub(runningBlock);
        }

        totalPopularlity = totalPopularlity.add(_horse.popularity);
    }

    return (normalizeReward, totalPopularlity);
}
```

5.8.2. Remediation

Since the `_horse.popularity` state is a dynamic state, Inspex suggests updating the `_horse.popularity` state in the `calculateRewardAndUpdateRemainHorse()` function via getting the state from the `Horse` contract.

Staking.sol

```
function calculateRewardAndUpdateRemainHorse(
    Horse[] storage _horses,
    uint256 _rewardPerAmount
) internal returns (uint256, uint256) {
    uint256 normalizeReward;
    uint256 totalPopularlity;

    for (uint256 index = 0; index < _horses.length; index++) {
        Horse storage _horse = _horses[index];

        uint256 runningBlock = block.number - _horse.lastRewardBlock; // change enterBlock to lastRewardBlock
        uint256 rewardPerBlock = _rewardPerAmount.div(runningBlock);

        if (runningBlock > _horse.remainBlock) {
            uint256 retriedBlock = runningBlock - _horse.remainBlock;
```

```
                normalizeReward = normalizeReward.add(
                    _horse.remainBlock.mul(rewardPerBlock).mul(
                        _horse.popularity
                    )
                );

                _horse.popularity = horse.getPopularity(_horse[tokenId]); // update popularity / 5

                _horse.remainBlock = 0; // update remainBlock from horse contract

                normalizeReward = normalizeReward.add(
                    retriedBlock.mul(rewardPerBlock).mul(_horse.popularity)
                );
            } else {
                normalizeReward = normalizeReward.add(
                    runningBlock.mul(rewardPerBlock).mul(_horse.popularity) // 2x
                );
                _horse.remainBlock = _horse.remainBlock.sub(runningBlock);
            }

            _horse.lastRewardBlock = block.number; // update every time
            totalPopularlity = totalPopularlity + _horse.popularity; // add new Popularity
        }

        return (normalizeReward, totalPopularlity);
    }
```

5.9. Improper runningBlock Value Calculation

ID	IDX-009
Target	Staking
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** High **Impact:** Medium Due to the miscalculation issue, the users will receive less reward than expected. When there is no reward in the contract, this also results in a denial of service on all deposit and withdraw functions. **Likelihood:** High This issue occurs whenever the user claims the reward or deposits/withdraws NFTs through the `Staking` contract.
Status	Resolved SpeedStar team has resolved this issue as suggested by implementing the `horse.lastRewardBlock` and updating its value every time that the user has claimed the reward in commit `3e39d7acf9c1aa9f3a5511c161c2035ba7d6bc1f`.

5.9.1. Description

The `horse.enterBlock` state will be assigned once while depositing to the Staking contract. For example in the `depositHorse()` function, the `horse.enterBlock` is assigned with `block.number` as shown in line 711.

Staking.sol

```
688 function depositHorse(uint256 _tokenId) external horseLimitStaking{
689     require(
690         horse.isApprovedForAll(msg.sender, address(this)),
691         "Please set approval"
692     );
693     PoolInfo storage pool = poolInfo;
694     UserInfo storage user = userInfo[msg.sender];
695     require(!user.ownedTokenId[_tokenId], "Already staking");
696 
697     updatePool();
698     if (user.amount > 0) {
699         payReward(user);
700     }
701 
702     uint256 popularity = horse.getPopularity(_tokenId);
703 
```

```
        horse.safeTransferFrom(address(msg.sender), address(this), _tokenId);
        user.ownedTokenId[_tokenId] = true;
        user.horseIndex[_tokenId] = user.horses.length;
        user.amount = user.amount.add(popularity);
        user.horses.push(
            Horse(
                _tokenId,
                block.number,
                horse.getRemainAge(_tokenId),
                popularity
            )
        );

        pool.totalStake = pool.totalStake.add(popularity);

        user.rewardDebt = user.amount.mul(pool.accSpeedPerShare).div(1e12);
        emit DepositHorse(msg.sender, popularity, _tokenId);
    }
```

In the `calculateRewardAndUpdateRemainHorse()` function, the `runningBlock` value is assigned in line 429 will always increase since the `_horse.enterBlock` has never been updated after the reward was claimed which affects to the reward calculation as in line 434, 441 and 447, resulting in reward miscalculation.

Staking.sol

```
struct Horse {
    uint256 tokenId;
    uint256 enterBlock;
    uint256 remainBlock;
    uint256 popularity;
}
```

Staking.sol

```
419 function calculateRewardAndUpdateRemainHorse(
420     Horse[] storage _horses,
421     uint256 _rewardPerAmount
422 ) internal returns (uint256, uint256) {
423     uint256 normalizeReward;
424     uint256 totalPopularlity;
425
426     for (uint256 index = 0; index < _horses.length; index++) {
427         Horse storage _horse = _horses[index];
428
429         uint256 runningBlock = block.number - _horse.enterBlock;
430         uint256 rewardPerBlock = _rewardPerAmount.div(runningBlock);
```

```
431         uint256 retriedReward = rewardPerBlock.div(5);
432
433         if (runningBlock > _horse.remainBlock) {
434             uint256 retriedBlock = runningBlock - _horse.remainBlock;
435             normalizeReward = normalizeReward.add(
436                 _horse.remainBlock.mul(rewardPerBlock).mul(
437                     _horse.popularity
438                 )
439             );
440             normalizeReward = normalizeReward.add(
441                 retriedBlock.mul(retriedReward).mul(_horse.popularity)
442             );
443             _horse.popularity = _horse.popularity.div(5);
444
445             _horse.remainBlock = 0;
446         } else {
447             normalizeReward = normalizeReward.add(
448                 runningBlock.mul(rewardPerBlock).mul(_horse.popularity) // 2x
449             );
450             _horse.remainBlock = _horse.remainBlock.sub(runningBlock);
451         }
452
453         totalPopularlity = totalPopularlity.add(_horse.popularity);
454     }
455
456     return (normalizeReward, totalPopularlity);
457 }
```

5.9.2. Remediation

Inspex suggests implementing the `horse.lastRewardBlock` and updating its value every time that the user has claimed the reward along with suggestion from the `IDX-008 Miscalculation in calculateRewardAndUpdateRemainHorse() Function` issue in lines 429 and 454, for example:

Staking.sol

```
struct Horse {
    uint256 tokenId;
    uint256 lastRewardBlock;
    uint256 remainBlock;
    uint256 popularity;
}
```

Staking.sol

```
419 function calculateRewardAndUpdateRemainHorse(
420     Horse[] storage _horses,
421     uint256 _rewardPerAmount
```

```
) internal returns (uint256, uint256) {
    uint256 normalizeReward;
    uint256 totalPopularlity;

    for (uint256 index = 0; index < _horses.length; index++) {
        Horse storage _horse = _horses[index];

        uint256 runningBlock = block.number - _horse.lastRewardBlock; // change enterBlock to lastRewardBlock
        uint256 rewardPerBlock = _rewardPerAmount.div(runningBlock);

        if (runningBlock > _horse.remainBlock) {
            uint256 retriedBlock = runningBlock - _horse.remainBlock;
            normalizeReward = normalizeReward.add(
                _horse.remainBlock.mul(rewardPerBlock).mul(
                    _horse.popularity
                )
            );

            _horse.popularity = horse.getPopularity(_horse[tokenId]); // update propularity / 5

            _horse.remainBlock = 0; // update remainBlock from horse contract

            normalizeReward = normalizeReward.add(
                retriedBlock.mul(rewardPerBlock).mul(_horse.popularity)
            );
        } else {
            normalizeReward = normalizeReward.add(
                runningBlock.mul(rewardPerBlock).mul(_horse.popularity) // 2x
            );
            _horse.remainBlock = _horse.remainBlock.sub(runningBlock);
        }

        _horse.lastRewardBlock = block.number; // update every time
        totalPopularlity = totalPopularlity + _horse.popularity; // add new Popularity
    }

    return (normalizeReward, totalPopularlity);
}
```

5.10. Centralized Control of State Variable

ID	IDX-010
Target	Staking JOC Speed Star Facility Horse Shop
Category	General Smart Contract Vulnerability
CWE	CWE-284: Improper Access Control
Risk	**Severity:** High **Impact:** High The controlling authorities can change the critical state variables to gain additional profit. Thus, it is unfair to the other users. **Likelihood:** Medium There is nothing to restrict the changes from being done; however, this action can only be done by the contract owner.
Status	Resolved * SpeedStar team has deployed the Timelock contract with a delay of 24 hours and transferred the ownership of affected contracts to the Timelock contract. The transfer ownership transactions are listed as follows: Staking : https://explorer.harmony.one/tx/0x931b4961dc9cd4b9ad4f2dda1d93f525aacb2bbcca54fc05a7539bad56783584 JOC : https://explorer.harmony.one/tx/0x5b599c59f5a35753d3c104190e73f408e604aa5218ffbdab5f9b76eb355ad980 Speed : https://explorer.harmony.one/tx/0x5abfd90b2f2bd67068620653104f121195c085b617e299b41978bfc46f7ded48 Star : https://explorer.harmony.one/tx/0x0e56af87c4066d994b68d7faa595e79a0042b402f1241c40774e1993805ad0cb Facility, Horse, Shop: The ownership will be transferred to the related contract only which contains the minting logic such as game rewarding, horse breeding, or facility shop.

5.10.1. Description

Critical state variables can be updated at any time by the controlling authorities. Changes in these variables can cause impacts to the users, so the users should accept or be notified before these changes are effective.

However, there is currently no constraint to prevent the authorities from modifying these variables without notifying the users.

The controllable privileged state update functions are as follows:

Target	Function	Modifier
Staking.sol(L:202)	setSpeedPerBlock()	onlyOwner
Staking.sol(L:235)	updateMultiplier()	onlyOwner
JOC.sol(L:902)	setAdmin()	onlyOwner
Speed.sol(L:902)	setAdmin()	onlyOwner
Star.sol(L:905)	setAdmin()	onlyOwner
Facility.sol(L:116)	setBaseURI()	onlyOwner
Horse.sol(L:111)	setBaseURI()	onlyOwner
Shop.sol(L:44)	setPriceFeed()	onlyOwner
Shop.sol(L:48)	setPackPrice()	onlyOwner
Shop.sol(L:53)	setPackAvaliable()	onlyOwner
Shop.sol(L:61)	setOpenSale()	onlyOwner

5.10.2. Remediation

In the ideal case, Inspex suggests removing the mentioned functions, the critical state variables should not be modifiable to keep the integrity of the smart contract. However, if modifications are needed, Inspex suggests limiting the use of these functions via the following options:

- Implementing a community-run governance to control the use of these functions
- Using a timelock mechanism to delay the changes for a reasonable amount of time, e.g., 24 hours

Please note that if the timelock mechanism is decided to be used, the minting functions in the `Facility` and `Horse` contracts will also be affected by the timelock. To avoid this issue all of the minting function modifiers must be changed, for example using `onlyMinter` modifier as follows:

Facility.sol

```
address minterAddress;
```

```
modifier onlyMinter() {
    require(msg.sender == minterAddress, "Not minter");
    _;
}

function setMinter(address _address) external onlyOwner {
    minterAddress = _address;
}
```

5.11. Improper claim() Function Implementation

ID	IDX-011
Target	Staking
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** High **Impact:** Medium When the `claim()` function is called, the user will not receive any reward and the user's reward will be marked as claimed. **Likelihood:** High This issue occurs every time when users call the `claim()` function to claim the reward.
Status	Resolved SpeedStar team has resolved this issue as suggested by updating the `poolInfo` and the `rewardDebt` state in commit `3e39d7acf9c1aa9f3a5511c161c2035ba7d6bc1f`.

5.11.1. Description

In the `Staking` contract, the `claim()` function is used for claiming the reward from the contract. However, There is no `updatePool()` function called before paying the reward and the user.`rewardDebt` state is updated before calling the `payReward()` function as shown in line 774.

Staking.sol

```
767 function claim() external {
768     PoolInfo storage pool = poolInfo;
769     UserInfo storage user = userInfo[msg.sender];
770     uint256 pending = user.amount.mul(pool.accSpeedPerShare).div(1e12).sub(
771         user.rewardDebt
772     );
773     if (pending > 0) {
774         user.rewardDebt = user.amount.mul(pool.accSpeedPerShare).div(1e12);
775         payReward(user);
776     }
777 }
```

5.11.2. Remediation

Inspex suggests calling the `updatePool()` function every time before paying the reward and calling the `payReward()` function before updating the `user.rewardDebt` as follows:

Staking.sol

```
function claim() external {
    PoolInfo storage pool = poolInfo;
    UserInfo storage user = userInfo[msg.sender];
    updatePool();
    uint256 pending = user.amount.mul(pool.accSpeedPerShare).div(1e12).sub(
        user.rewardDebt
    );
    if (pending > 0) {
        payReward(user);
        user.rewardDebt = user.amount.mul(pool.accSpeedPerShare).div(1e12);
    }
}
```

5.12. Missing stable.multiplier Multiplication in Reward Calculation

ID	IDX-012
Target	Staking
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** High **Impact:** Medium All users will receive a smaller amount of reward from staked horses in the stable than expected due to the miscalculation issue. **Likelihood:** High All users who staked horses in the stable will be affected.
Status	Resolved SpeedStar team has resolved this issue as suggested by multiplication the `horseReward` state by the `stable.multiplier` state in commit `3e39d7acf9c1aa9f3a5511c161c2035ba7d6bc1f`.

5.12.1. Description

According to the business design, the reward from staking can be boosted by staking the horse into the stable, so that the reward calculation will take place by multiplying the `horse.popularity` with the `stable.multiplier`.

In the `pendingSpeed()` function, the reward from the staked horses in the stable is added to the `normalizeReward` in line 319 which is not multiplied with the `stable.multiplier`:

Staking.sol

```
272 function pendingSpeed(address _user) external view returns (uint256) {
273     PoolInfo storage pool = poolInfo;
274     UserInfo storage user = userInfo[_user];
275     uint256 accSpeedPerShare = pool.accSpeedPerShare;
276     uint256 lpSupply = pool.totalStake;
277 
278     if (block.number > pool.lastRewardBlock && lpSupply != 0) {
279         uint256 multiplier = getMultiplier(
280             pool.lastRewardBlock,
281             block.number
282         );
283         uint256 wagReward = multiplier
284             .mul(speedPerBlock)
285             .mul(pool.allocPoint)
```

```
            .div(totalAllocPoint);
        accSpeedPerShare = accSpeedPerShare.add(
            wagReward.mul(1e12).div(lpSupply)
        );
    }

    uint256 pending = user.amount.mul(accSpeedPerShare).div(1e12).sub(
        user.rewardDebt
    );

    if (pending > 0) {
        uint256 rewardPerAmount = pending.div(user.amount);
        uint256 normalizeReward;

        // calculate horse reward and popularlity
        (uint256 reward, ) = calculateReward(user.horses, rewardPerAmount);

        normalizeReward = normalizeReward.add(reward);

        for (uint256 index = 0; index < user.facility.length; index++) {
            normalizeReward = normalizeReward.add(
                IFacility(facility).popularity(user.facility[index]).mul(
                    rewardPerAmount
                )
            );
        }
        // calculate reward and update populality;
        for (uint256 index = 0; index < user.stables.length; index++) {
            Stable storage stable = user.stables[index];
            (uint256 horseReward, ) = calculateReward(
                stable.horses,
                rewardPerAmount
            );
            normalizeReward = normalizeReward.add(horseReward);
        }

        return normalizeReward;
    } else {
        return 0;
    }
}
```

In the `payReward()` function, the reward from the staked horses in the stable is added to the `normalizeReward` in line 370 which is not multiplied with the `stable.multiplier`:

Staking.sol

```
328 function payReward(UserInfo storage _user) internal {
```

```
    uint256 pending = _user
        .amount
        .mul(poolInfo.accSpeedPerShare)
        .div(1e12)
        .sub(_user.rewardDebt);

    if (pending > 0) {
        uint256 rewardPerAmount = pending.div(_user.amount);
        uint256 normalizeReward;
        uint256 newPopularity;
        // calculate horse reward and popularlity
        (
            uint256 reward,
            uint256 popularlity
        ) = calculateRewardAndUpdateRemainHorse(
                _user.horses,
                rewardPerAmount
            );

        normalizeReward = normalizeReward.add(reward);
        newPopularity = popularlity;

        for (uint256 index = 0; index < _user.facility.length; index++) {
            uint256 facilityPopularlity = IFacility(facility).popularity(
                _user.facility[index]
            );
            normalizeReward = normalizeReward.add(
                facilityPopularlity.mul(rewardPerAmount)
            );
            newPopularity = newPopularity.add(facilityPopularlity);
        }
        // calculate reward and update populality;
        for (uint256 index = 0; index < _user.stables.length; index++) {
            Stable storage stable = _user.stables[index];
            (
                uint256 horseReward,
                uint256 totalPopularlity
            ) = calculateRewardAndUpdateRemainHorse(
                    stable.horses,
                    rewardPerAmount
                );
            normalizeReward = normalizeReward.add(horseReward);
            stable.popularity = totalPopularlity.mul(stable.multiplier);
            newPopularity = newPopularity.add(stable.popularity);
        }

        require(
```

```
                newPopularity <= pending,
                "normalizeReward is not over pending reward."
            );
            _user.amount = newPopularity;
            safeSpeedTransfer(msg.sender, normalizeReward);
        }
}
```

5.12.2. Remediation

Inspex suggests multiplying the `horseReward` with `stable.multiplier` before adding it to the `normalizeReward` state.

Staking.sol

```
    // calculate reward and update popularity;
    for (uint256 index = 0; index < user.stables.length; index++) {
        Stable storage stable = user.stables[index];
        (uint256 horseReward, ) = calculateReward(
            stable.horses,
            rewardPerAmount
        );
        normalizeReward =
normalizeReward.add(horseReward.mul(stable.multiplier));
    }
```

Staking.sol

```
    // calculate reward and update popularity;
    for (uint256 index = 0; index < _user.stables.length; index++) {
        Stable storage stable = _user.stables[index];
        (
            uint256 horseReward,
            uint256 totalPopularlity
        ) = calculateRewardAndUpdateRemainHorse(
                stable.horses,
                rewardPerAmount
            );
        normalizeReward =
normalizeReward.add(horseReward.mul(stable.multiplier));
        stable.popularity = totalPopularlity.mul(stable.multiplier);
        newPopularity = newPopularity.add(stable.popularity);
    }
```

5.13. Missing Native Token Withdrawal Function

ID	IDX-013
Target	Staking
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Medium **Impact:** Low The `claimToken()` function is required user to transfer gas fee to the `Staking` contract, but the contract does not have any function to claim the gas fee which lead to the gas fee is stuck in the contract and the platform must pay **Likelihood:** High It is likely to happen since the `claimToken()` can be called by anyone. Also, this is a majority function the users will interact with.
Status	Resolved SpeedStar team has resolved this issue as suggested by adding the `claimNativeToken()` function to withdraw the native token from the contract in commit `3e39d7acf9c1aa9f3a5511c161c2035ba7d6bc1f`.

5.13.1. Description

The `claimToken()` function is used in the `Staking` contract to claim tokens from off-chain. With the current design, the user transfers the native token to the contract for off-chain gas, which then transfers the token to the user, as shown below:

```
// For user claim token from offchain
// Reserve gas for offchain call to deposit token to user.
function claimToken(address _token) external payable {
    require(msg.value == 50000000 gwei, ""); //reserve 0.05 one for backend
fee.
    emit ClaimToken(msg.sender, _token);
}
```

Furthermore, in the `Staking` contract, it does not have the function for withdrawing native tokens from the contract. Hence, the gas will be stuck in the contract forever.

5.13.2. Remediation

Since the `Staking` contract does not allow any depositing of any native token except through the `claimToken()` function, implementing the function to transfer the native token from the contract will not affect the user fund.

Inspex suggests adding the function to withdraw the native token from `Staking` contract, which allows only the administrator role. For example:

```
// For user claim token from offchain
// Reserve gas for offchain call to deposit token to user.
function claimNativeToken() external onlyOwner {
    (bool sent, bytes memory data) = msg.sender.call{value: address(this).balance}("");
    require(sent, "Failed to send Native Token");
    emit ClaimToken(msg.sender, _token);
}
```

5.14. Improper horseLimitStaking() Function Implementation

ID	IDX-014
Target	Staking
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Medium **Impact:** Low Due to the miscalculation of the `starBalance` state and missing update of the `totalHorse` state, the user can stake more than two horses for free which violates the business design as it requires the user to stake the $STAR in order to get more slots. **Likelihood:** High The issue is likely that users can stake more than two horses per wallet. Thus, the affected function can be called without any restriction.
Status	Resolved SpeedStar team has resolved this issue as suggested by changing the implementation of the `horseLimitStaking` modifier and updating the `userInfo[msg.sender].totalHorse` state in commit `3e39d7acf9c1aa9f3a5511c161c2035ba7d6bc1f`.

5.14.1. Description

The `horseLimitStaking()` modifier limits that the users can deposit only two horses for free. If the users want to stake more than two horses, the users must stake $STAR to get more slots.

However, the implementation of the `horseLimitStaking()` modifier now is shown below.

Staking.sol

```
163 modifier horseLimitStaking() {
164     // start 2 slots to free stake. after that increase follower Star staking
165     require(
166         starBalance[msg.sender].add(2) > userInfo[msg.sender].totalHorse,
167         "Slot not enough."
168     );
169     _;
170 }
```

In the previous source code in line 166, the `starBalance` state is used for recording the staking $STAR balance. Since the $STAR decimals is 18, the user can stake only 1 $STAR and the `starBalance` will be set to 1,000,000,000,000,000,000. As a result, the user can stake the huge amount of horse after stake for just 1 $STAR.

Moreover in the previous source code as in line 166, the `userInfo[msg.sender].totalHorse` state is not updated properly. For example in the `depositHorse()` function, there is no update part for the `userInfo[msg.sender].totalHorse` state.

Staking.sol

```
function depositHorse(uint256 _tokenId) external horseLimitStaking{
    require(
        horse.isApprovedForAll(msg.sender, address(this)),
        "Please set approval"
    );
    PoolInfo storage pool = poolInfo;
    UserInfo storage user = userInfo[msg.sender];
    require(!user.ownedTokenId[_tokenId], "Already staking");

    updatePool();
    if (user.amount > 0) {
        payReward(user);
    }

    uint256 popularity = horse.getPopularity(_tokenId);

    horse.safeTransferFrom(address(msg.sender), address(this), _tokenId);
    user.ownedTokenId[_tokenId] = true;
    user.horseIndex[_tokenId] = user.horses.length;
    user.amount = user.amount.add(popularity);
    user.horses.push(
        Horse(
            _tokenId,
            block.number,
            horse.getRemainAge(_tokenId),
            popularity
        )
    );

    pool.totalStake = pool.totalStake.add(popularity);

    user.rewardDebt = user.amount.mul(pool.accSpeedPerShare).div(1e12);
    emit DepositHorse(msg.sender, popularity, _tokenId);
}
```

5.14.2. Remediation

Inspex suggests calculating the decimals of the `starBalance` state and updating the `userInfo[msg.sender].totalHorse` state when user deposits or withdraws the horse. For example:

Staking.sol

```
modifier horseLimitStaking() {
    // start 2 slots to free stake. after that increase follower Star staking
    require(
starBalance[msg.sender].add(2000000000000000000).div(1000000000000000000) > userInfo[msg.sender].totalHorse,
        "Slot not enough."
    );
    _;
}
```

Staking.sol

```
function depositHorseInStable(uint256 _stableTokenId, uint256 _horseTokenId)
    external horseLimitStaking
{
    PoolInfo storage pool = poolInfo;
    UserInfo storage user = userInfo[msg.sender];

    require(user.ownedStable[_stableTokenId], "No stable staking");
    userInfo[msg.sender].totalHorse = userInfo[msg.sender].totalHorse.add(1);

    updatePool();
    if (user.amount > 0) {
        payReward(user);
    }
```

Staking.sol

```
function withdrawHorseInStable(
    uint256 _stableTokenId,
    uint256 _horseTokenId
) public {
    PoolInfo storage pool = poolInfo;
    UserInfo storage user = userInfo[msg.sender];
    require(user.ownedStable[_stableTokenId], "No stable staking");
    userInfo[msg.sender].totalHorse = userInfo[msg.sender].totalHorse.sub(1);

    updatePool();
    uint256 pending = user.amount.mul(pool.accSpeedPerShare).div(1e12).sub(
        user.rewardDebt
    );
```

Staking.sol

```
function depositHorse(uint256 _tokenId) external horseLimitStaking{
    require(
        horse.isApprovedForAll(msg.sender, address(this)),
        "Please set approval"
    );
    PoolInfo storage pool = poolInfo;
    UserInfo storage user = userInfo[msg.sender];
    require(!user.ownedTokenId[_tokenId], "Already staking");
    userInfo[msg.sender].totalHorse = userInfo[msg.sender].totalHorse.add(1);

    updatePool();
    if (user.amount > 0) {
        payReward(user);
    }
```

Staking.sol

```
function withdrawHorse(uint256 _tokenId) external {
    PoolInfo storage pool = poolInfo;
    UserInfo storage user = userInfo[msg.sender];
    require(user.ownedTokenId[_tokenId], "withdraw: not good");
    userInfo[msg.sender].totalHorse = userInfo[msg.sender].totalHorse.sub(1);

    updatePool();
    uint256 pending = user.amount.mul(pool.accSpeedPerShare).div(1e12).sub(
        user.rewardDebt
    );
```

5.15. Incorrect Price Incremental Calculation in buyPack() Function

ID	IDX-015
Target	Shop
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Medium **Impact:** Low Normally, the `packPriceDollar[0]` state should be increased every 100 sales of the `_packId` with value 0 (stable pack). However, due to the incorrect price increment, the `packPriceDollar[0]` state will be increased by 10 every sale until `packAvaliable[0]` state is below 100. **Likelihood:** High The `packPriceDollar[0]` state will be incorrectly increased every time the users buy packs through the `buyPack()` function.
Status	Resolved SpeedStar team has resolved this issue as suggested by changing the implementation of the `buyPack()` function in commit `3e39d7acf9c1aa9f3a5511c161c2035ba7d6bc1f`.

5.15.1. Description

In the `buyPack()` function, the `packPriceDollar[0]` state will be increased every time when `packAvaliable[0].div(100)` is not equal to 0 as shown in line 94:

Shop.sol

```
81 function buyPack(uint16 _packId) public payable {
82     require(openSale, "Not open sale");
83     require(packPriceDollar[_packId] > 0, "Price not set");
84     require(packAvaliable[_packId] > 0, "Not avaliable");
85     uint256 rate = getONERate();
86     require(rate != 0, "Not found rate for swap.");
87 
88     uint256 payAmount = getPackPrice(_packId);
89     require(msg.value >= payAmount, "pay amount mismatch");
90 
91     packAvaliable[_packId] = packAvaliable[_packId].sub(1);
92     // each 100 stable to selled the price is increase to 10$
93     if (_packId == 0) {
94         if (packAvaliable[_packId].div(100) != 0) {
95             packPriceDollar[_packId] = packPriceDollar[_packId].add(10);
96         }
```

```
    }
    emit BuyPack(_packId, payAmount, msg.sender);
}
```

This mean every `_packId` 0 sales before the last 100 packs will increase the `packPriceDollar[0]` by 10

5.15.2. Remediation

For increasing the price every 100 sales, Inspex suggests using the modulo operator (%) and checking that the result is equal to 0 as shown in line 94:

Shop.sol

```
81 function buyPack(uint16 _packId) public payable {
82     require(openSale, "Not open sale");
83     require(packPriceDollar[_packId] > 0, "Price not set");
84     require(packAvaliable[_packId] > 0, "Not avaliable");
85     uint256 rate = getONERate();
86     require(rate != 0, "Not found rate for swap.");
87
88     uint256 payAmount = getPackPrice(_packId);
89     require(msg.value >= payAmount, "pay amount mismatch");
90
91     packAvaliable[_packId] = packAvaliable[_packId].sub(1);
92     // each 100 stable to selled the price is increase to 10$
93     if (_packId == 0) {
94         if (packAvaliable[_packId] % (100) == 0) {
95             packPriceDollar[_packId] = packPriceDollar[_packId].add(10);
96         }
97     }
98     emit BuyPack(_packId, payAmount, msg.sender);
99 }
```

5.16. Improper Sale Properties Modification During On-Going Sale Event

ID	IDX-016
Target	Shop
Category	Advanced Smart Contract Vulnerability
CWE	CWE-284: Improper Access Control
Risk	**Severity:** Medium **Impact:** Medium The modification of the sale properties is unfair for the users since the total number of packs and the price can be changed from what is known by the users. This results in loss of reputation for the platform and monetary impact for the users. **Likelihood:** Medium Only the owner can modify the states, and there is a benefit for the owner in performing this action, so there is a motivation for the attack.
Status	Resolved SpeedStar team has resolved this issue as suggested by validating the pack state after open sale in commit `3e39d7acf9c1aa9f3a5511c161c2035ba7d6bc1f`.

5.16.1. Description

In the Shop contract the `setPriceFeed()`, `setPackPrice()` and `setPackAvaliable()` functions can be called by the owner and change the state at any time. If this action is maliciously done during the sale, the users can unknowingly buy a pack with an exceedingly high price.

Shop.sol

```
function setPriceFeed(address _address) public onlyOwner {
    onePriceFeed = AggregatorV3Interface(_address);
}

function setPackPrice(uint16 _packId, uint256 _price) external onlyOwner {
    packPriceDollar[_packId] = _price;
    emit SetPackPrice(_packId, _price);
}

function setPackAvaliable(uint16 _packId, uint256 _amount)
    external
    onlyOwner
{
    packAvaliable[_packId] = _amount;
    emit SetPackAvaliable(_packId, _amount);
}
```

5.16.2. Remediation

Inspex suggests adding conditions to prevent these functions from being used during an on-going sale event, for example:

Shop.sol

```
function setPriceFeed(address _address) public onlyOwner {
    require(!openSale, "Unable to set during sale");
    onePriceFeed = AggregatorV3Interface(_address);
}

function setPackPrice(uint16 _packId, uint256 _price) external onlyOwner {
    require(!openSale, "Unable to set during sale");
    packPriceDollar[_packId] = _price;
    emit SetPackPrice(_packId, _price);
}

function setPackAvaliable(uint16 _packId, uint256 _amount)
    external
    onlyOwner
{
    require(!openSale, "Unable to set during sale");
    packAvaliable[_packId] = _amount;
    emit SetPackAvaliable(_packId, _amount);
}
```

5.17. Loop Over Unbounded Data Structure

ID	IDX-017
Target	Staking
Category	General Smart Contract Vulnerability
CWE	CWE-400: Uncontrolled Resource Consumption
Risk	**Severity:** Low **Impact:** Medium The affected functions will eventually be unusable due to excessive gas usage. **Likelihood:** Low It is very unlikely that the `horses`, the `facility`, and the `stable` state sizes will be raised until the affected function is eventually unusable.
Status	Resolved SpeedStar team has resolved this issue as suggested by implementing the plot limit (total size of staked facilities) and `emergencyWithdrawHorse()` function in commit `3e39d7acf9c1aa9f3a5511c161c2035ba7d6bc1f`.

5.17.1. Description

In the `getPopularityInStable()` function, the source code loops through the `user.stables` state to find the desired index of the `user.stables` as shown in line 467.

Staking.sol

```
459 function getPopularityInStable(uint256 _stableId)
460     public
461     view
462     returns (uint256)
463 {
464     UserInfo storage user = userInfo[msg.sender];
465     uint256 popularity;
466     for (uint256 index = 0; index < user.stables.length; index++) {
467         if (user.stables[index].tokenId == _stableId) {
468             for (
469                 uint256 j = 0;
470                 j < user.stables[index].horses.length;
471                 j++
472             ) {
473                 popularity = popularity.add(
474                     user.stables[index].horses[j].popularity
475                 );
476             }
```

```
            popularity = popularity.mul(user.stables[index].multiplier);

            break;
        }
    }

    return popularity;
}
```

With the current design, there is no limit amount of stable that a user can deposit. If a user deposits the stable, the `user.stables.length` will continue to grow and this function will eventually be unusable due to excessive gas usage.

Since the stable can be withdrawn with the `withdrawStable()` function, it will call the `withdrawHorseInStable()` function which calls the `getPopularityInStable()` function anyway.

As a result, this issue leads to Denial-of-Service in the `withdrawHorseInStable()` and the `withdrawStable()` functions. There are also other affected functions as in the following table:

Target	Contract	Function
Staking.sol (L: 272)	Staking	pendingSpeed()
Staking.sol (L: 328)	Staking	payReward()
Staking.sol (L: 384)	Staking	calculateReward()
Staking.sol (L: 419)	Staking	calculateRewardAndUpdateRemainHorse()
Staking.sol (L: 459)	Staking	getPopularityInStable()

5.17.2. Remediation

Inspex suggests adding the mechanism to validate the bound of the `horses`, the `facility`, and the `stable` states in the affected function.

In some cases, it can be resolved by changing the algorithm of the contract instead of looping through the unbound structure.

For example, changing the algorithm to find the index of stable by referencing from the `user.stableIndex` state, which records the index of stable in the `user.stables` state.

Staking.sol

```
function getPopularityInStable(uint256 _stableId)
    public
    view
    returns (uint256)
{

    UserInfo storage user = userInfo[msg.sender];
    uint256 popularity;

    for (
        uint256 j = 0;
        j < user.stables[user.stableIndex[_stableId]].horses.length;
        j++
    ) {
        popularity = popularity.add(
            user.stables[user.stableIndex[_stableId]].horses[j].popularity
        );
    }
    popularity =
popularity.mul(user.stables[user.stableIndex[_stableId]].multiplier);


    return popularity;
}
```

5.18. Insufficient Logging for Privileged Functions

ID	IDX-018
Target	Staking JOC Speed Star Facility Horse Shop
Category	General Smart Contract Vulnerability
CWE	CWE-778: Insufficient Logging
Risk	**Severity:** Very Low **Impact:** Low Privileged functions' executions cannot be monitored easily by the users. **Likelihood:** Low It is not likely that the execution of the privileged functions will be a malicious action.
Status	Resolved SpeedStar team has resolved this issue as suggested by emitting events for the execution of privileged functions in commit `3e39d7acf9c1aa9f3a5511c161c2035ba7d6bc1f`.

5.18.1. Description

Privileged functions that are executable by the controlling parties are not logged properly by emitting events. Without events, it is not easy for the public to monitor the execution of those privileged functions, allowing the controlling parties to perform actions that cause big impacts on the platform.

For example, the owner can set the `BONUS_MULTIPLIER` state by executing the `updateMultiplier()` function in the `Staking` contract, and no event is emitted.

Staking.sol

```
235 function updateMultiplier(uint256 multiplierNumber) external onlyOwner {
236     BONUS_MULTIPLIER = multiplierNumber;
237 }
```

The following table shows the privileged functions without any event emitted:

Target	Function	Modifier
Staking.sol(L:235)	updateMultiplier()	onlyOwner

JOC.sol(L:902)	setAdmin()	onlyOwner
Speed.sol(L:902)	setAdmin()	onlyOwner
Star.sol(L:905)	setAdmin()	onlyOwner
Facility.sol(L:116)	setBaseURI()	onlyOwner
Horse.sol(L:83)	setAge()	onlyOwner
Horse.sol(L:111)	setBaseURI()	onlyOwner
Shop.sol(L:44)	setPriceFeed()	onlyOwner
Shop.sol(L:118)	claimToken()	onlyOwner

5.18.2. Remediation

Inspex suggests emitting events for the execution of privileged functions, for example:

Staking.sol

```
event UpdateMultiplier(uint256 multiplierNumber);
function updateMultiplier(uint256 multiplierNumber) external onlyOwner {
    BONUS_MULTIPLIER = multiplierNumber;
    emit UpdateMultiplier(multiplierNumber);
}
```

5.19. Inexplicit Solidity Compiler Version

ID	IDX-019
Target	Staking JOC Speed Star Facility Horse Shop
Category	Smart Contract Best Practice
CWE	CWE-1104: Use of Unmaintained Third Party Components
Risk	**Severity:** Info **Impact: None** **Likelihood: None**
Status	Resolved SpeedStar team has resolved this issue as suggested by fixing the Solidity compiler to the latest stable version in commit `3e39d7acf9c1aa9f3a5511c161c2035ba7d6bc1f` .

5.19.1. Description

The Solidity compiler versions declared in the smart contracts were not explicit. Each compilation may be done using different compiler versions, which may potentially result in compatibility issues.

Contract Name	Version
Staking	^0.8.0
JOC	^0.8.0
Speed	^0.8.0
Star	^0.8.0
Facility	^0.8.0
Horse	^0.8.0
Shop	^0.8.0

5.19.2. Remediation

Inspex suggests fixing the Solidity compiler to the latest stable version. At the time of the audit, the latest stable version of Solidity compiler in major 0.8 is v0.8.12 [4].

5.20. Improper Function Visibility

ID	IDX-020
Target	Facility Horse Shop
Category	Smart Contract Best Practice
CWE	CWE-710: Improper Adherence to Coding Standards
Risk	**Severity:** Info **Impact: None** **Likelihood: None**
Status	Resolved SpeedStar team has resolved this issue as suggested by changing the visibility to external in commit `3e39d7acf9c1aa9f3a5511c161c2035ba7d6bc1f`.

5.20.1. Description

Functions with public visibility copy calldata to memory when being executed, while external functions can read directly from calldata. Memory allocation uses more resources (gas) than reading directly from calldata.

The following source code shows that the `setAge()` function of the `Horse` contract is set to public and it is never called from any internal function.

Horse.sol

```
function setAge(uint256 _tokenId, uint256 _age) public onlyOwner {
    age[_tokenId] = _age;
}
```

The following table contains all functions that have public visibility and are never called from any internal function.

Target	Function
Facility(L:55)	mintStables()
Horse(L:83)	setAge()
Horse(L:87)	getRemainAge()
Shop(L:44)	setPriceFeed()

Shop(L:118)	claimToken()

5.20.2. Remediation

In this case, Inspex suggests changing all functions' visibility to external if they are not called from any internal function as shown in the following example:

Horse.sol

```
function setAge(uint256 _tokenId, uint256 _age) external onlyOwner {
    age[_tokenId] = _age;
}
```

5.21. Incorrect Logging Parameter

ID	IDX-021
Target	Staking
Category	Smart Contract Best Practice
CWE	CWE-710: Improper Adherence to Coding Standards
Risk	**Severity:** Info **Impact: None** **Likelihood: None**
Status	Resolved SpeedStar team has resolved this issue as suggested by emitting the facility's popularity in commit `3e39d7acf9c1aa9f3a5511c161c2035ba7d6bc1f`.

5.21.1. Description

The facility's popularity should be emitted in the `DepositFacility` event as shown in line 132:

Staking.sol

```
130 event DepositFacility(
131     address indexed user,
132     uint256 popularity,
133     uint256 tokenId
134 );
```

However, the facility's popularity is not emitted correctly. The 0 popularity value is emitted instead in the `DepositFacility` event as shown in line 505:

Staking.sol

```
486 function depositFacility(uint256 _tokenId) external {
487     UserInfo storage user = userInfo[msg.sender];
488     require(!user.ownedFacility[_tokenId], "Already staking");
489 
490     updatePool();
491     if (user.amount > 0) {
492         payReward(user);
493     }
494 
495     facility.safeTransferFrom(address(msg.sender), address(this), _tokenId);
496 
497     // update amount
498     uint256 popularity = facility.popularity(_tokenId);
```

```
    user.amount = user.amount.add(popularity);
    user.ownedFacility[_tokenId] = true;

    user.facilityIndex[_tokenId] = user.facility.length;
    user.facility.push(_tokenId);

    emit DepositFacility(msg.sender, 0, _tokenId);
}
```

Furthermore, the **popularity** value in the **DepositStable** event will not be able to be emitted since the calling functions are not able to provide the **popularity** value for emitting.

Staking.sol

```
event DepositStable(
    address indexed user,
    uint256 popularity,
    uint256 tokenId
);
```

Staking.sol

```
function depositStable(uint256 _tokenId) external {
    UserInfo storage user = userInfo[msg.sender];
    require(!user.ownedStable[_tokenId], "Already staking");

    updatePool();
    if (user.amount > 0) {
        payReward(user);
    }
    uint256 multiplier = facility.multipliers(_tokenId);
    facility.safeTransferFrom(address(msg.sender), address(this), _tokenId);

    Stable[] storage userStable = user.stables;
    user.stableIndex[_tokenId] = userStable.length;
    user.ownedStable[_tokenId] = true;
    userStable.push();

    Stable storage newStable = userStable[user.stableIndex[_tokenId]];
    newStable.tokenId = _tokenId;
    newStable.multiplier = multiplier;

    emit DepositStable(msg.sender, 0, _tokenId);
}
```

5.21.2. Remediation

Inspex suggests emitting the facility's popularity as shown in line 505:

Staking.sol

```
486 function depositFacility(uint256 _tokenId) external {
487     UserInfo storage user = userInfo[msg.sender];
488     require(!user.ownedFacility[_tokenId], "Already staking");
489 
490     updatePool();
491     if (user.amount > 0) {
492         payReward(user);
493     }
494 
495     facility.safeTransferFrom(address(msg.sender), address(this), _tokenId);
496 
497     // update amount
498     uint256 popularity = facility.popularity(_tokenId);
499     user.amount = user.amount.add(popularity);
500     user.ownedFacility[_tokenId] = true;
501 
502     user.facilityIndex[_tokenId] = user.facility.length;
503     user.facility.push(_tokenId);
504 
505     emit DepositFacility(msg.sender, popularity, _tokenId);
506 }
```

Inspex also recommends removing the `popularity` value from the `DepositStable` event and updating the emit parameter in line 561 as follows:

Staking.sol

```
event DepositStable(
    address indexed user,
    uint256 tokenId
);
```

Staking.sol

```
541 function depositStable(uint256 _tokenId) external {
542     UserInfo storage user = userInfo[msg.sender];
543     require(!user.ownedStable[_tokenId], "Already staking");
544 
545     updatePool();
546     if (user.amount > 0) {
547         payReward(user);
548     }
549     uint256 multiplier = facility.multipliers(_tokenId);
```

```
    facility.safeTransferFrom(address(msg.sender), address(this), _tokenId);

    Stable[] storage userStable = user.stables;
    user.stableIndex[_tokenId] = userStable.length;
    user.ownedStable[_tokenId] = true;
    userStable.push();

    Stable storage newStable = userStable[user.stableIndex[_tokenId]];
    newStable.tokenId = _tokenId;
    newStable.multiplier = multiplier;

    emit DepositStable(msg.sender, _tokenId);
}
```

5.22. Use of transfer() Function to Transfer Native Token

ID	IDX-022
Target	Shop
Category	Smart Contract Best Practice
CWE	CWE-710: Improper Adherence to Coding Standards
Risk	**Severity:** Info **Impact: None** **Likelihood: None**
Status	Resolved SpeedStar team has resolved this issue as suggested by using the `call()` function to transfer native tokens in commit `3e39d7acf9c1aa9f3a5511c161c2035ba7d6bc1f`.

5.22.1. Description

Using the `transfer()` and `send()` functions for sending native tokens might result in the transaction reverted since the amount of forwarded gas is limited at 2300.

In the Shop contract, the native token is transferred with the `transfer()` function as shown in line 119:

Shop.sol

```
118 function claimToken() public onlyOwner {
119     payable(msg.sender).transfer(address(this).balance);
120 }
```

5.22.2. Remediation

Inspex suggests using the `call()` function to transfer native token instead as follows:

Shop.sol

```
function claimToken() public onlyOwner {
    (bool sent, bytes memory data) = msg.sender.call{value:
address(this).balance}("");
    require(sent, "Failed to send Ether");
}
```

6. Appendix

6.1. About Inspex

CYBERSECURITY
PROFESSIONAL
SERVICE

Inspex is formed by a team of cybersecurity experts highly experienced in various fields of cybersecurity. We provide blockchain and smart contract professional services at the highest quality to enhance the security of our clients and the overall blockchain ecosystem.

Follow Us On:

Website	https://inspex.co
Twitter	@InspexCo
Facebook	https://www.facebook.com/InspexCo
Telegram	@inspex_announcement

6.2. References

[1] "OWASP Risk Rating Methodology." [Online]. Available: https://owasp.org/www-community/OWASP_Risk_Rating_Methodology. [Accessed: 25-February-2022]

[2] "Harmony VRF" [Online]. Available: https://docs.harmony.one/home/developers/tools/harmony-vrf. [Accessed: 25-February-2022]

[3] "ReentrancyGuard" [Online]. Available: https://docs.openzeppelin.com/contracts/4.x/api/security#ReentrancyGuard. [Accessed: 25-February-2022]

[4] "Version 0.8.12" [Online]. Available: https://github.com/ethereum/solidity/releases/tag/v0.8.12. [Accessed: 25-February-2022]

inspex

CYBERSECURITY PROFESSIONAL SERVICE