Smart Contract Audit Report

Prepared by: **Cybring**
Prepared for: **Starship**

Version: 1.0 (Final report)
Date: 17th February 2025
Commit/Hash: 5dfe6b91dfd785d458dd32bf5b0814a3ec89f658
Auditor: Anh Bui
Reviewer: Duc Cuong Nguyen

Contents

1 Introduction

1.1 Project overview

`Starship` is a community-driven multichain fundraising platform designed to offer the community a fair chance to access potential high-quality projects while serving as a reliable bridge between investors and truly long-term builders.

1.2 Objective

This security audit report was provided by `CYBRING` on January 22, 2025. This audit was conducted to assess the robustness, reliability, and security of the code base. The goal was to identify vulnerabilities, ensure compliance with best security practices, and provide mitigation measures.

After the initial audit report, a reassessment was conducted on 17th February 2025 to verify the status of the reported issues. While some issues were addressed, others were acknowledged and will be prioritized for future improvement.

1.3 Disclaimer

This security audit is not produced to replace any other type of assessment and does not aim to guarantee the discovery of all security issues within the scope of the assessment. While this audit was carried out with good faith and technical proficiency, it's crucial to understand that no single audit can guarantee the absolute security of a smart contract. To minimize risks, `CYBRING` recommends a multi-faceted approach involving multiple independent assessments. Please note that this report is not intended to provide financial advice.

2 Scope of Audit

The final audit assessment was conducted on commit 5dfe6b91dfd785d458dd32bf5b0814a3ec89f658. Further details regarding the `Starship` audit scope are provided below:

- **Smart contracts audited:** See Appendix.
- **Codebase details:**
  - Language: Solidity
  - Frameworks: Hardhat
  - Category: Launchpad
  - Chain: Viction
  - Initial audit's commit hash: 0ca4ca0e703a9366c0cd7447bc23dd6bcfd0e91d
  - Reassessment audit's commit hash: 5dfe6b91dfd785d458dd32bf5b0814a3ec89f658
- **Deploying state:** Table 1 shows the current address of the smart contract deployed on VIC SCAN.

Deploying Smart Contract	Address
StarshipFactory	0xe5c93231e4659dd0d0dc6cb44157e28e6d906d6a
StarshipSaleStandard	0xf90e776ed25d3b0a672429d3a19d613db5297697
Broadcaster	0x927637d4997e24944952d8d6609ec184694f8dd6
Forwarder	0x1c33d1183a429b6ce9b50fc35a812644e58a889d

Table 1: Deploying smart contract address

3 Audit Summary

Severity	Count
Medium	1
Low	2
Informational	4

4 Methodology

CYBRING conducts the following procedures to enhance security level of our client's smart contracts:

- **Pre-auditing:** Understanding the business logic of the smart contracts, investigating the deployment states of samples, and preparing for the audit.
- **Auditing:** Examining the smart contract by evaluating on multiple perspectives:
  - **Manual code review:** CYBRING auditors evaluate the static code analysis report for finding false positive reports. Besides, inspect the smart contract logic, access controls, and data flows ensure the contract behaves as intended and is free from risky logic.
  - **Static code analysis:** By using advanced static analysis techniques combined with our customized detectors, we set out to identify potential vulnerabilities, optimize gas usage, and ensure adherence to best practices in smart contract development.
  - **Fuzz testing:** CYBRING leverages fuzzing tools to stress-test the contract, ensuring it performs securely under unpredictable conditions.
- **First deliverable and consulting:** Presenting an initial report on the findings with recommendations for remediation and offering consultation services.
- **Reassessment:** Verifying the status of the issues and identifying any additional complications in the implemented fixes.
- **Final deliverable:** Delivering a comprehensive report detailing the status of each issue.

4.1 Audit categories

- **Common vulnerabilities:** Smart contracts are analyzed following OWASP smart contract top 10 and Smart Contract Weakness Classification (SWC).
- **Advanced vulnerabilities:** `CYBRING` simulates a certain types of attack scenarios to exploit the smart contracts. These scenarios were prioritized from high to low severity.
- **Security best practices:** The source code of the smart contract is analyzed from the development perspective, providing suggestions for improving the overall code quality.

4.2 Audit items

Table 2 show the details of the issues our auditors will conduct the audit upon.

Category	**Item**
Reentrancy attack	Reentrancy attack Reentrancy via modifier Cross function reentrancy Cross contract reentracy
Data Validation and Integrity	Integer overflow and underflow Missing zero address validation Builtin symbols, state variables, and function should not be shadowed Storage variables should be initialized at the time of declaration
Denial of service (DOS)	Denial of service with revert Denial of service with gas limit Denial of service with induction variable overflow Denial of service by exceeding the maximum call stack depth
Access control vulnerabilities	Unprotected access to call sensitive function (self-destruct/suicide) Dangerous usage of tx.origin
Private information and randomness	Store sensitive information on smart contracts Generate random numbers from insecure pseudorandom factors
Timestamp dependence	Timestamp dependence on critical function
Best practices	Missing event for changing critical access control Follow standard style guide (naming conversion, code layout, order of layout) from Solidity

Table 2: Details of examined items

4.3 Risk rating

The OWASP Risk Rating Methodology was applied to assess the severity of each issue based on the following criteria, arranged from the perspective of smart contract security.

- **Likelihood:** a measure of how likely this vulnerability is to be discovered and exploited by an attacker.
- **Impact:** a measure of the potential consequences or severity of a vulnerability if it is exploited by an attacker, including the extent of damage, data loss, or disruption of operations.

Impact				
	High	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Informational	Low	Medium
		Low	*Medium*	*High*
		Likelihood		

Table 3: Overall Risk Severity

5 Detailed Findings

5.1 Unused constant that is inherited from an abstract contract

- **Description:** The constant *REWARD_PRECISION* is defined as an attribute of the abstract smart contract *StarshipPool*. However, it is used only in the derived smart contract *StarshipPoolFarm*. Therefore, including *REWARD_PRECISION* in the abstract smart contract is unnecessary.
- **Risk:** Informational
- **CWE:** CWE-398: Indicator of Poor Code Quality
- **Affected code:**

```
11      abstract contract StarshipPool is StarshipCore, IStarshipPool {
12          using SafeERC20 for IERC20;
13
14          uint256 constant REWARD_PRECISION = 1 ether;
```

contracts/StarshipPool.sol:11-14

- **Mitigation:** To avoid potential unintended problems, it is recommended to move the constant to the derived smart contract.
- **Status [17/02/2025]:** The `Starship` team has acknowledged this issue.

5.2 Local variables shadowing

- **Description:** The *owner* parameter in the *createPad()* function shadows the *owner()* function defined in the *Ownable* abstract smart contract. This shadowing creates ambiguity, as the parameter name conflicts with the function name, potentially leading to confusion.
- **Risk:** Low
- **CWE:**
  - CWE-833: Dead Code
  - CWE-710: Improper Adherence to Coding Standards
- **Affected code:**

```
107     function createPad(address owner, uint256 protocolFee, uint
            lockTime, uint256 typeID, bytes32 salt) external onlyAdmin
            returns (address pad) {
```

contracts/StarshipFactory.sol:107

- **Mitigation:** To resolve this, the parameter name should be changed to prevent overlap with the function name.
- **Status [17/02/2025]:** The `Starship` team has acknowledged this issue.

5.3 Missing zero address validation

- **Description:** In Solidity, missing zero address validation refers to the failure to check whether an address parameter is the zero address before using it in a function or assigning it to a critical role in a smart contract.
- **Risk:** Informational
- **CWE:** CWE-20: Improper Input Validation
- **Affected code:**

```
57      function redeem(bytes32 dID, address exchangeToken, uint256
            launchpadTokenAmount, bytes memory extra) external virtual
             override payable {
58        if (!_registeredDIDs[dID]) {
59          _register(dID, extra);
60        }
61
62        _redeem(dID, exchangeToken, launchpadTokenAmount);
63      }
```

contracts/StarshipSale.sol:57-63

```
34    function getQuoteForRedeem(bytes32, address exchangeToken,
          uint256 launchpadTokenAmount) external override view returns
          (uint256) {
35      ExchangeInfo memory exchangeInfo = _getExchangeInfo(
            exchangeToken);
```

```
36        require(exchangeInfo.priceN > 0, "Launchpad: Exchange is not
              active");
37
38        return launchpadTokenAmount * exchangeInfo.priceN /
              exchangeInfo.priceD;
39    }
```

contracts/StarshipSale.sol:34-39

```
61    function getQuoteForRedeem(bytes32 dID, address exchangeToken,
          uint256 launchpadTokenAmount) external override view returns
          (uint256) {
62        if(_userShares[dID] == 0) {
63            ExchangeInfo memory exchangeInfo = _getExchangeInfo(
                  exchangeToken);
64            require(exchangeInfo.priceN > 0, "Launchpad: Exchange is
                  not active");
65
66            return launchpadTokenAmount * exchangeInfo.priceN /
                  exchangeInfo.priceD;
67        }
68        return 0;
69    }
```

contracts/StarshipPool.sol:61-69

- **Mitigation:** To resolve this, zero address validation should be implemented before using the address variable.
- **Status [17/02/2025]:** The `Starship` team has acknowledged this issue. After thorough analysis and testing under the contract's actual deployment and usage conditions, we have verified and concluded that the circumstances necessary to exploit this issue cannot arise in practice. We have changed the corresponding category to Informational.

5.4 Redundant virtual modifier

- **Description:** In `Starship` smart contracts, there are derived contracts that inheritance *virtual* methods without overriding them.
- **Risk:** Informational
- **CWE:**
  - CWE-710: Improper Adherence to Coding Standards
  - CWE-682: Incorrect Calculation
- **Affected code:**
  - Missing overriding *register()* function in StarshipPoolSimple and StarshipPoolFarm
  - Missing overriding *redeem()* function in StarshipPoolSimple and StarshipPoolFarm

  - Missing overriding *setExchange()* function in StarshipPoolSimple, StarshipPoolFarm, StarshipSaleSequenced, and StarshipSaleStandard

- **Mitigation:** To resolve this, remove *virtual* modifier of the following function:
  - *register()* at StarshipPool.sol:76-78
  - *redeem()* at StarshipPool.sol:99-105
  - *setExchange()* at StarshipPool.sol:120-128
  - *setExchange()* at StarshipSale.sol:89-96
- **Status [17/02/2025]:** The `Starship` team has acknowledged this issue.

5.5 Not applying SafeMath in VRC25

- **Description:** Certain operations in the contract involving VRC25 tokens do not utilize SafeMath or equivalent mechanisms to handle arithmetic operations safely. Failure to apply SafeMath in token operations may lead to unexpected behavior, such as incorrect balances, fund losses, or security vulnerabilities.
- **Risk:** Informational
- **Affected code:** VRC25.sol
- **Mitigation:** To resolve this, apply SafeMath on VRC25 token for preventing arithmetic vulnerabilities.
- **Status [17/02/2025]:** The `Starship` team has acknowledged this issue.

5.6 Using deprecated functions

- **Description:** The *safeApprove()* function has been marked as deprecated by OpenZeppelin since 2020. This function not only includes several potential security issues but also incurs unnecessary gas usage.
- **CWE:**
  - CWE-477: Use of Obsolete Functions
  - CWE-1104: Use of Unmaintained Third-Party Components
  - CWE-937: Use of Components with Known Vulnerabilities
- **Risk:** Low
- **Affected code:**

```
26      function commit(address starship, bytes32 dID, address
            exchangeToken, uint256 exchangeTokenAmount, bytes memory
            extra) external payable {
27          if (exchangeToken != address(0)) {
28              IERC20(exchangeToken).safeTransferFrom(msg.sender,
                    address(this), exchangeTokenAmount);
29              IERC20(exchangeToken).safeApprove(starship,
                    exchangeTokenAmount);
```

```
30          }
31
32          IStarshipPool(starship).commit{value: msg.value}(dID,
                exchangeToken, exchangeTokenAmount, extra);
33      }
```

contracts/Forwarder.sol:26-33

```
35      function redeem(address starship, bytes32 dID, address
            exchangeToken, uint256 launchpadTokenAmount, bytes memory
            extra) external payable {
36          uint256 quote = IStarshipCore(starship).getQuoteForRedeem(dID
                , exchangeToken, launchpadTokenAmount);
37          if (exchangeToken != address(0)) {
38              IERC20(exchangeToken).safeTransferFrom(msg.sender, address(
                    this), quote);
39              IERC20(exchangeToken).safeApprove(starship, quote);
40          }
41
42          IStarshipCore(starship).redeem{value: msg.value}(dID,
                exchangeToken, launchpadTokenAmount, extra);
43      }
```

contracts/Forwarder.sol:35-43

- **Mitigation:** To mitigate the use of the deprecated *safeApprove()* function from the SafeERC20 library, *safeIncreaseAllowance()* and *safeDecreaseAllowance()* functions should be used instead.
- **Status [17/02/2025]:** The `Starship` team has addressed this issue, following `CYBRING`'s suggestion.

5.7 Insecure Logic in Time Lock Mechanism

- **Description:** The TimeLock contract is implemented to use functions within a pre-defined time interval. However, the current implementation includes a security vulnerability. The lock management variable *_isUnlock* does not reset to *False* after the *locktime* has passed.

  Scenario of a security incident: When the owner unlocks a critical function but does not use it, malicious miners can alter the timestamp for accessing the restricted function.
- **CWE:** CWE-664: Improper Control of a Resource Through Its Lifetime
- **Risk:** Medium
- **Affected code:** VRC25.sol, TimeLock.sol
- **Mitigation:** To resolve this, `CYBRING` suggests resetting the lock management to *False* to lock all functions after the interval time.
- **Status [17/02/2025]:** The `Starship` team has addressed this issue.

Appendix

```
contracts/
├── interfaces/
│   ├── IBroadcaster.sol
│   ├── IStarshipCore.sol
│   ├── IStarshipFactory.sol
│   ├── IStarshipPool.sol
│   ├── IStarshipSale.sol
│   └── IVRC25.sol
├── libraries/
│   ├── AccessControl.sol
│   ├── BroadcasterHelper.sol
│   ├── FreeManager.sol
│   ├── Payable.sol
│   ├── ProxyFactory.sol
│   ├── TimeLock.sol
│   ├── VRC25.sol
│   └── Whitelist.sol
├── Forwarder.sol
├── StarshipCore.sol
├── StarshipFactory.sol
├── StarshipPool.sol
├── StarshipPoolFarm.sol
├── StarshipPoolSimple.sol
├── StarshipSale.sol
├── StarshipSaleSequenced.sol
└── StarshipSaleStandard.sol
```