verichains

SECURITY AUDIT OF

SKYARK CHRONICLES BIDDING SMART CONTRACT

Public Report

Jan 26, 2024

Verichains Lab

info@verichains.io

https://www.verichains.io

Driving Technology > Forward

ABBREVIATIONS

Name	Description
Ethereum	An open source platform based on blockchain technology to create and distribute smart contracts and decentralized applications.
Ether (ETH)	A cryptocurrency whose blockchain is generated by the Ethereum platform. Ether is used for payment of transactions and computing services in the Ethereum network.
Smart contract	A computer protocol intended to digitally facilitate, verify or enforce the negotiation or performance of a contract.
Solidity	A contract-oriented, high-level language for implementing smart contracts for the Ethereum platform.
Solc	A compiler for Solidity.

EXECUTIVE SUMMARY

This Security Audit Report was prepared by Verichains Lab on Jan 26, 2024. We would like to thank the SkyArk for trusting Verichains Lab in auditing smart contracts. Delivering high-quality audits is always our top priority.

This audit focused on identifying security flaws in code and the design of the SkyArk Chronicles Bidding smart contract. The scope of the audit is limited to the source code files provided to Verichains. Verichains Lab completed the assessment using manual, static, and dynamic analysis techniques.

During the audit process, the audit team found no vulnerability in the given version of SkyArk Chronicles Bidding smart contract, only some notes and recommendations.

TABLE OF CONTENTS

1. MANAGEMENT SUMMARY

1.1. About SkyArk Chronicles Bidding smart contract

SkyArk is a Triple-A game creation system and online gaming platform backed by Binance Labs. SkyArk Universe is a platform that supports multiple game titles and offer NFT interoperability across titles.

1.2. Audit scope

This audit focused on identifying security flaws in code and the design of the smart contracts of SkyArk Chronicles Bidding smart contract. It was conducted on the source code provided by the SkyArk team.

The following file was made available in the course of the review:

SHA256 Sum	File
a70c84b64cb8bf4758d75eb639e8ed28640ffb4231a0c5836299db8d0f04b68f	bidding.sol

1.3. Audit methodology

Our security audit process for smart contract includes two steps:

- Smart contract codes are scanned/tested for commonly known and more specific vulnerabilities using public and RK87, our in-house smart contract security analysis tool.
- Manual audit of the codes for security issues. The contracts are manually analyzed to look for any potential problems.

Following is the list of commonly known vulnerabilities that were considered during the audit of the smart contract:

- Integer Overflow and Underflow
- Timestamp Dependence
- Race Conditions
- Transaction-Ordering Dependence
- DoS with (Unexpected) revert
- DoS with Block Gas Limit
- Gas Usage, Gas Limit and Loops
- Redundant fallback function
- Unsafe type Inference
- Reentrancy
- Explicit visibility of functions state variables (external, internal, private and public)

- Logic Flaws

For vulnerabilities, we categorize the findings into categories as listed in table below, depending on their severity level:

SEVERITY LEVEL	DESCRIPTION
CRITICAL	A vulnerability that can disrupt the contract functioning; creates a critical risk to the contract; required to be fixed immediately.
HIGH	A vulnerability that could affect the desired outcome of executing the contract with high impact; needs to be fixed with high priority.
MEDIUM	A vulnerability that could affect the desired outcome of executing the contract with medium impact in a specific scenario; needs to be fixed.
LOW	An issue that does not have a significant impact, can be considered as less important.

Table 1. Severity levels

1.4. Disclaimer

SkyArk acknowledges that the security services provided by Verichains, are conducted to the best of their professional abilities but cannot guarantee 100% coverage of all security vulnerabilities. SkyArk understands and accepts that despite rigorous auditing, certain vulnerabilities may remain undetected. Therefore, SkyArk agrees that Verichains shall not be held responsible or liable, and shall not be charged for any hacking incidents that occur due to security vulnerabilities not identified during the audit process.

1.5. Acceptance Minute

This final report served by Verichains to the SkyArk will be considered an Acceptance Minute. Within 7 days, if no any further responses or reports is received from the SkyArk, the final report will be considered fully accepted by the SkyArk without the signature.

2. AUDIT RESULT

2.1. Overview

The SkyArk Chronicles Bidding smart contract was written in `Solidity` language, with the required version to be `^0.8.20`. The source code was written based on OpenZeppelin's library.

The SkyArk Chronicles Bidding smart contract is a bidding contract which let users bid for SKYARK Genesis. The bidding process is divided into 2 round: `Spellbook` and `Public`.

The `Public` round is open to all users. Bids can only be submitted within the specified timeframe and the price must be between specified price range set by the contract owner. Multiple bid submissions are allowed but the amount must be higher than the old one. The latest bid will be considered, and the amount from the previous bid will be automatically refunded. The contract owner can let users to `quit` bidding by providing them valid signatures. Users can then use the signature to withdraw their bids.

The `Spellbook` round is exclusive to `Spellbook` users and the price will be fixed. The contract owner need to provide `Spellbook` users valid signature to join this round.

The contract owner can modify price, time range and withdraw deposited funds at any time.

2.2. Findings

During the audit process, the audit team found no vulnerability in the given version of SkyArk Chronicles Bidding smart contract, only some notes and recommendations.

#	Issue	Severity
1	There may not be enough money for users to quit bidding	INFORMATIVE
2	Should enforce min/max value to avoid misconfiguration	INFORMATIVE
3	Increase amount instead of refund to reduce gas cost	INFORMATIVE

2.2.1. There may not be enough money for users to quit bidding INFORMATIVE

The contract let the contract owner withdraws any amount of fund at any time so if the owner withdraws more than the necessary, users will be unable to `quit` and claim their refund when they lost the bidding.

```
function quit(bytes memory signature) public {
    address recoverAddress = _recoverSignatureFromParam(
        quitSelector,
        signature
    );
```

```
    require(recoverAddress == operator, "Incorrect signature");
    uint256 refundAmount = userWithPublicBid[msg.sender];
    require(refundAmount > 0, "No refundable funds");
    userWithPublicBid[msg.sender] = 0;
    payable(msg.sender).transfer(refundAmount);
    emit QuitBid(msg.sender, block.timestamp, refundAmount);
}

function withdraw(address to, uint256 amount) public onlyOwner {
    require(address(to) != address(0), "Cannot withdraw to 0 address");
    require(address(this).balance >= amount, "Insufficient funds");
    payable(to).transfer(amount);
}
```

RECOMMENDATION

Add a `receive` function to let the contract owner deposit fund back to refund the users.

2.2.2. Should enforce min/max value to avoid misconfiguration INFORMATIVE

The contract should enforce `_minPrice` < `_maxPrice` and `_startTime` < `_endTime` in modify functions to avoid misconfiguration.

```
function modifySpellbookBidTime(
    uint256 _startTime,
    uint256 _endTime
) public onlyOwner {
    spellbookBidding.startTime = _startTime;
    spellbookBidding.endTime = _endTime;
}

function modifyPublicBidTime(
    uint256 _startTime,
    uint256 _endTime
) public onlyOwner {
    publicBidding.startTime = _startTime;
    publicBidding.endTime = _endTime;
}

function modifyPublicBidPrice(
    uint256 _minPrice,
    uint256 _maxPrice
) public onlyOwner {
    publicBidding.minPrice = _minPrice;
    publicBidding.maxPrice = _maxPrice;
}
```

2.2.3. Increase amount instead of refund to reduce gas cost INFORMATIVE

The contract can increase `userWithPublicBid` by additional `msg.value` instead of receive new amount and refunding the previous bid amount to reduce the gas cost.

```
function bidForPublic() public payable onlyWhenBidForPublicStart {
    uint256 _currentBidPrice = userWithPublicBid[msg.sender];
    require(
        msg.value > _currentBidPrice &&
        msg.value >= publicBidding.minPrice &&
        msg.value <= publicBidding.maxPrice,
        "Invalid bid"
    );
    if (_currentBidPrice > 0) {
        payable(msg.sender).transfer(_currentBidPrice);
        emit Refund(msg.sender, block.timestamp, _currentBidPrice);
    }
    userWithPublicBid[msg.sender] = msg.value;
    emit PublicBid(msg.sender, block.timestamp, msg.value);
}
```

3. VERSION HISTORY

Version	Date	Status/Change	Created by
1.0	*Jan 26, 2024*	Public Report	Verichains Lab

Table 2. Report versions history