Delabs Games
Boxing Star X Wallet (Kaia)

Security
Audit
Report

February 7, 2025

1. INTRODUCTION

1.1 Disclaimer

The audit makes no statements or warranties about utility of the code, safety of the code, suitability of the business model, investment advice, endorsement of the platform or its products, regulatory regime for the business model, or any other statements about fitness of the contracts to purpose, or their bug free status. The audit documentation is for discussion purposes only. The information presented in this report is confidential and privileged. If you are reading this report, you agree to keep it confidential, not to copy, disclose or disseminate without the agreement of the Client. If you are not the intended recipient(s) of this document, please note that any disclosure, copying or dissemination of its content is strictly forbidden.

1.2 Security Assessment Methodology

BugBlow utilized a widely adopted approach to performing a security audit. Below is a breakout of how our team was able to identify and exploit vulnerabilities.

1.2.1 Code and architecture review:

- Review the source code.
- Read the project's documentation.
- Review the project’s architecture.

Stage goals

- Build a good understanding of how the application works.

1.2.2 Check code against vulnerability checklist:

- Understand the project’s critical assets, resources, and security requirements.
- Manual check for vulnerabilities based on the auditor's checklist.
- Run static analyzers.

Stage goals

- Identify and eliminate typical vulnerabilities (gas limit, replay, flash-loans attack, etc.)

1.2.3 Threat & Attack Simulation:

- Analyze the project's critical assets, resources and security requirements.
- Exploit the found weaknesses in a safe local environment.
- Document the performed work.

Stage goals

- Identify vulnerabilities that are not listed in static analyzers that would likely be exploited by hackers.
- Develop Proof of Concept (PoC).

1.2.4 Report found vulnerabilities:

- Discuss the found issues with developers
- Show remediations.
- Write an initial audit report.

Stage goals

- Verify that all found issues are relevant.

1.2.5 Fix bugs & re-audit code:

- Help developers fix bugs.
- Re-audit the updated code again.

Stage goals

- Double-check that the found vulnerabilities or bugs were fixed and were fixed correctly.

1.2.4. Deliver final report:

- The Client deploys the re-audited version of the code
- The final report is issued for the Client.

Stage goals

- Provide the Client with the final report that serves as security proof.

Severity classification

All vulnerabilities discovered during the audit are classified based on their potential severity and have the following classification:

Severity	Description
Critical	Vulnerabilities leading to assets theft, fund access locking, or any other loss of funds.
High	Vulnerabilities that can trigger a contract failure. Further recovery is possible only by manual modification of the contract state or replacement.
Medium	Vulnerabilities that can break the intended contract logic or expose it to DoS attacks, but do not cause direct loss funds.
Low	Vulnerabilities that do not have a significant immediate impact and could be easily fixed.

1.3 Project Overview

BSX Wallet allows players to purchase in-game items using both native coins (Kaia) and ERC-20 tokens. The project also supports airdrops and claims, allowing the developers to distribute rewards to players conveniently.

1.4 Project Dashboard

Project Summary

Title	Description
Client name	Delabs Games
Project name	Boxing Star X Wallet
Timeline	04.02.2025 - 07.02.2025
Number of Auditors	3

Project Last Log

Date	Data Hash	Note
02.02.2025	Tx: 0xdf9d70535e0c2452d000e4e98d04cbb8bf2169dba14a9091a85833736ed5d7a6	

Project Scope

The audit covered the following files:

File name	Link
contracts/BSXWallet.sol	https://kairos.kaiascope.com/account/0x9c9cb6fe233bf5e1f13c4b4ba36990d42df5c97e?tabId=contractCode

1.5 Summary of findings

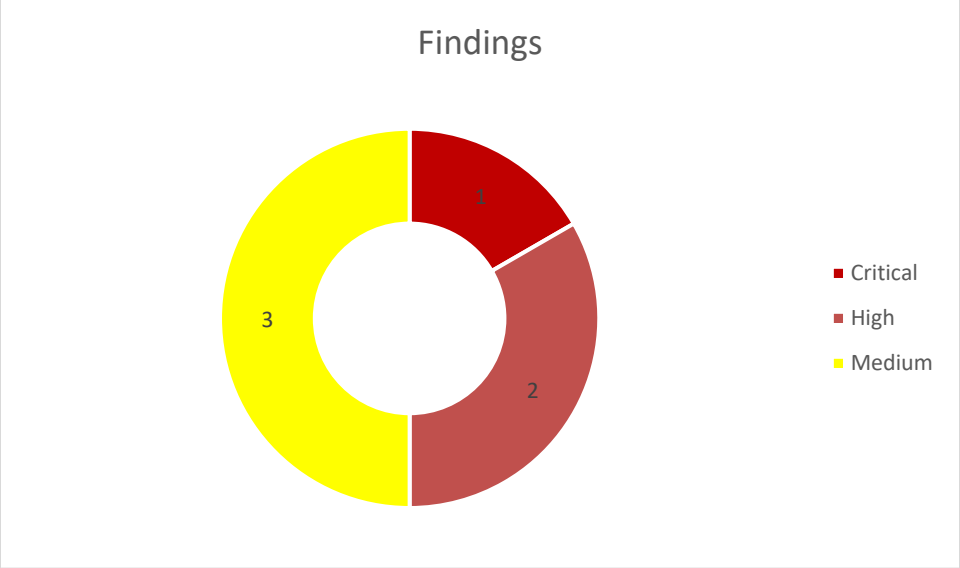

Figure 1. Findings chart

Severity	# of Findings
CRITICAL	1
HIGH	2
MEDIUM	3
LOW	0

2. FINDINGS REPORT

2.1 Critical

2.1.1 Signature Replay Vulnerability

Status: <?>

Description

Functions claimToken() and claimKAIA() verify the fields according to the signature. However, the contract never increments or changes userNonce[msg.sender]. This means that, once collected, the same signature can be reused indefinitely to claim tokens or native coin repeatedly, **draining the contract's balance**. It is especially critical for claimKAIA() function because anybody can call it and receive money on their account.

```
function claimKAIA(uint256 _amount, bytes memory _sig) external nonReentrant{
    bytes32 message = prefixed(keccak256(abi.encodePacked(msg.sender, _amount, chainId, this,
userNonce[msg.sender])));
    require(recoverSigner(message, _sig) == signVerifier, "Invalid Signature");

    require(address(this).balance >= _amount, "Insufficient funds");

    (bool sent, ) = msg.sender.call{value: _amount}("");
    require(sent, "Failed to send Ether");

    emit ClaimKAIA(msg.sender, _amount);
}
```

Remediation

There are many ways how a nonce can be changed. Assuming that the signer application just increments it, the contract should change it as well after signature validation. A simple example:

```
userNonce[msg.sender] += 1;
```

In this case, the contract should have a getLastNonce(address) method, so that the signer application can keep up with what nonce it should send in the signature. Also, please keep in mind that the contract can't increase userNonce for forever, there are integer limits. The best way to solve it would be to combine a *nonce* and an expiration date (*timestamp*).

For example:

The *nonce* can change within 0-1000 range, but to invalidate very old signatures (the previous 0-1000 bulk), the signer could also send an expiration date in the signature and to the contract (i.e. 1 hour), so that the contract could compare it to *block.timestamp* and reject old signatures.

```
keccak256(abi.encodePacked(msg.sender, _amount, deadline, userNonce[msg.sender]));
```

2.2 High

2.2.1 Items price manipulation

Status: Fixed

Description

Anyone from the whitelist can set the price for an item. Apart from getting the game items for free, this can be especially dangerous if you implement the refund logic, because there might be vulnerabilities in the future so that malicious actors would be able to refund more than they should.

```
function setTokenPrice(address[] calldata _tokenAddresses, uint256[] calldata _price) external only-
WhiteList{
    require( _tokenAddresses.length == _price.length, "Length wrong");

    for( uint256 i = 0; i < _tokenAddresses.length; i++ ){
        tokenPrice[_tokenAddresses[i]] = _price[i];
    }
}
```

Remediation

Please consider limiting the number of users/entities who can set the price (only owner) or using an Oracle in the future.

2.2.2 Buying unavailable items.

Status: Fixed

Description

Missing itemActive check in buyItemKAIA(). The function does not confirm *itemActive[_itemId].* By contrast, buyItem() enforces this policy.

This means a user can buy a coin-based item even if the item is unavailable. In addition, this can cause inconsistency in the off-chain backend storage and cause unexpected behavior.

Remediation

Insert a line that checks whether the item is active.

```
function buyItemKAIA(uint256 _itemId) payable external nonReentrant {
    require(isSalesActive, "Not Activated");
    require(isCoinActive, "Not support");
    require(itemActive[_itemId], "Item is not active"); // Enforce item is active

    uint256 amount = itemPrices[_itemId];
    require(amount <= msg.value, "Insufficient funds");

    // ...
}
```

2.3 Medium

2.3.1 No refund on overpayment.

Status: Fixed

Description

If the buyer sends more coins than necessary, the contract silently keeps the excess.

This doesn't protect the buyer from making a mistake.

Remediation

You can add a refund function that would return the excess back to the sender. (Please also mark it as non-reentrant, private)

```
uint256 needed = itemPrices[_itemId];
require(msg.value >= needed, "Insufficient funds");
uint256 overpay = msg.value - needed;
if (overpay > 0) {
    (bool refundSuccess, ) = msg.sender.call{value: overpay}("");
    require(refundSuccess, "Refund failed");
}
```

2.3.2 Single point of trust

Status: Acknowledged

Description

Currently, critical administrative tasks (such as withdrawing funds, setting prices, or activating items) rely on a single owner's address. If that wallet's private key is compromised or lost, an attacker could gain full control over the contract.

Remediation

Consider implement a multi-signature (multi-sig) wallet in the near future for administrative functions. A multi-sig setup typically requires confirmation from multiple authorized addresses (e.g., M-of-N signatures) before executing privileged operations. This ensures that a single compromised key cannot compromise the entire contract.

2.3.3 Denial of service with large maps

Status: Fixed

Description

Functions buyItems, airdropBatchPayment, airdropBatchPaymentKAIA, setItemPrices, changeItemsActive, setTokenPrice, setTokenPrice iterate over unlimited maps. If the maps are large enough, it may exceed the gas limit in a single block, effectively rendering the functions unavailable.

Remediation

For the moment, consider limiting the number of map records to a certain number. For the future, consider using the factory pattern where for every purchase, a contract is created and removed later [1].

2.3 Low

Not found

CONSCLUSION

The BSXWallet smart contract provides essential functionalities (buying items with tokens or native coins, handling airdrops, and allowing claims via off-chain signatures). However, it presents multiple vulnerabilities—including 1 Critical, 3 High, and 3 Medium issues— that require immediate attention:

1. Critical:

   • *Signature Replay*: claimToken() and claimKAIA() do not update userNonce, enabling repeated usage of a single signature to drain contract funds.
2. High:

   • Items Price Manipulation: Whitelisted addresses can set item prices at will, creating a risk of free or underpriced items (especially dangerous if a future refund mechanism is introduced).

   • Unprotected claimKAIA(): claimToken() is restricted to onlyOwner, but claimKAIA() is not. This design gap, combined with the replay vulnerability, allows anyone to exploit signatures.

   • Purchasing Unavailable Items: buyItemKAIA() does not check itemActive[_itemId], letting users buy inactive or unavailable items.
3. Medium:

   • No Refund on Overpayment: Buyers who send too much native coin receive no refund, risking accidental loss of funds.

   • Single Point of Trust (No Multi-Signature): Critical administrative actions rely on a single owner key, creating a high risk if that key is compromised or lost.

   • Denial of Service with Large Maps: Functions like buyItems, airdropBatchPayment, setItemPrices can iterate over arbitrarily large arrays, potentially exceeding gas limits and causing unavailability.

Addressing the issues outlined will significantly enhance the reliability, fairness, and usability of the Boxing Star X Wallet smart contract. We advise implementing the recommended changes and conducting a follow-up audit to ensure that no new issues are introduced.

REFERENCES

[1] https://medium.com/better-programming/learn-solidity-the-factory-pattern-75d11c3e7d29

[2] https://chain.link/education-hub/replay-attack