Delabs Games
Boxing Star X Wallet

Security
Audit
Report

July 16, 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.

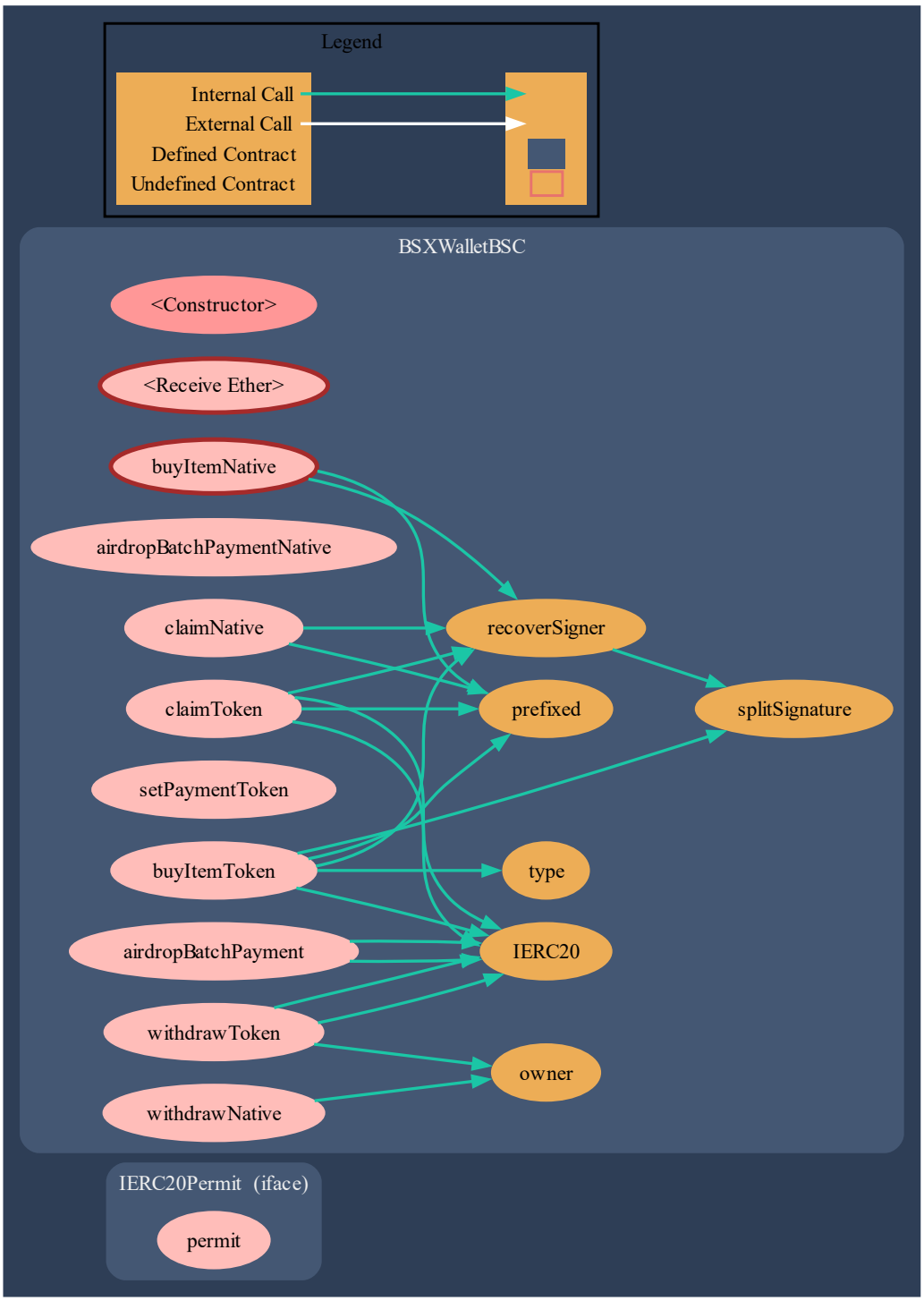

Figure 1. Contracts graph flow.

1.4 Project Dashboard

Project Summary

Title	Description
Client name	Delabs Games
Project name	Boxing Star X Wallet
Timeline	07.04.2025 - 07.14.2025
Number of Auditors	3

Project Scope

The audit covered the following files:

File name
contracts/BSXWalletBSC.sol contracts/BSXWalletKAIA.sol

1.5 Summary of findings

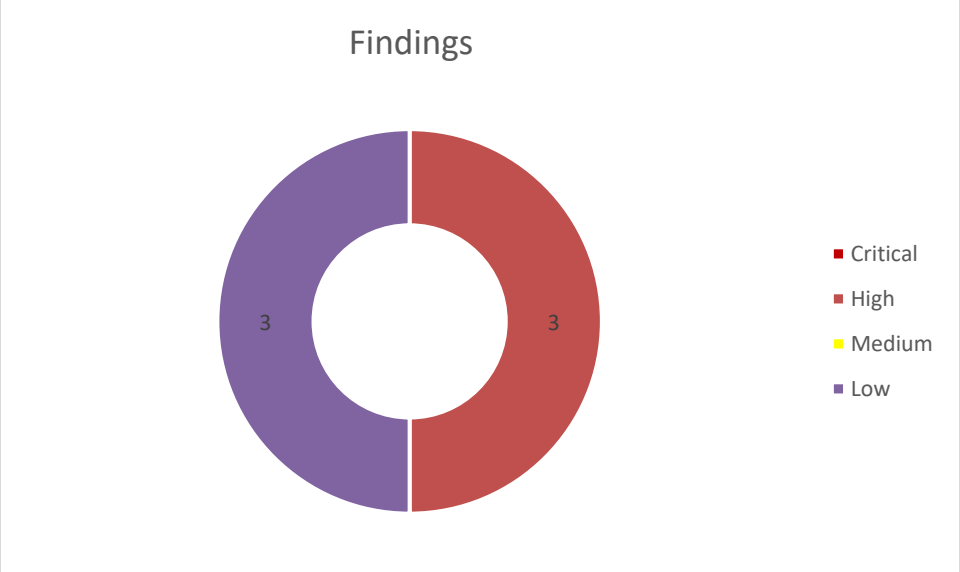

Figure 2. Findings chart

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

2. FINDINGS REPORT

2.1 Critical

Not Found

2.2 High

2.2.1 Excessive Permit Allowance

Status: Acknowledged

Description

Using type(uint256).max in permit() grants unlimited allowance, which, if the contract or its owner is compromised, allows attackers to drain *all* tokens from any user who interacted with `buyItemToken`. If control over the contract's owner key is gained by an attacker (e.g., via a social engineering attack on your team members), the consequences would be: fund loss for users and reputational damage.

```
IERC20Permit(tokenAddress).permit(
    msg.sender,
    address(this),
    type(uint256).max,
    p.deadline,
    v,
    s
);
```

Remediation

Limit the approved allowance to the actual amount (p.amount) being spent.

```
IERC20Permit(tokenAddress).permit(
    msg.sender,
    address(this),
    p.amount,
    p.deadline,
    v,
    s
);
```

2.2.2 Unbounded Loop May Exceed Block Gas Limit in Airdrops

Status: Fixed

Description

Large input arrays can cause the airdrop transaction to exceed the block gas limit and revert. This represents a direct financial loss for the owner with no successful outcome. The caller (the contract owner) will pay for all the gas consumed up to the point of the revert. The Ethereum Virtual Machine (EVM) has a maximum gas limit per block (currently around 30 million gas). Any transaction that attempts to consume more gas than this limit will fail and be reverted. Both `airdropBatchPayment` and `airdropBatchPaymentNative` iterate through arrays (_walletAddresses and _amounts), performing a transfer for each recipient in a for loop.

Remediation

Add a constant batch size limit (e.g., AIRDROP_BATCH_LIMIT) and enforce it with require. Alternatively, replace it with a pull-based "claim" model for scalable airdrops.

2.2.3 DoS Risk via Reverting Recipient in Airdrop Loop

Status: Acknowledged

Description

A single malicious recipient address reverting on transfer() or call{value: ...}("") can cause DoS on the entire airdrop and waste your funds for gas. Furthermore, unless such an actor is found out, it can render next airdrops unsuccessful as well, if not excluded from the lists.

```
contract AirdropRecipient {
    // This fallback will revert any ETH sent to the contract
    receive() external payable {
        revert("Rejecting native transfers");
    }
    // This function will revert if tokens are transferred using ERC-20
    fallback() external payable {
        revert("Rejecting token transfers");
    }
}
```

Remediation

Preferred: Implement a pull-based airdrop where users call claim() to receive funds individually.

Alternative: Wrap each transfer in a try/catch (for call) or low-level call() and handle failures with logging,

```
1. (bool sent, ) = walletAddress.call{value: amount}("");
2. if (!sent) {
3.     emit AirdropFailed(walletAddress, amount);
4.     continue;
5. }
```

Another Alternative: skip unsuccessful transfers.

2.3 Medium

Not Found

2.4 Low

2.4.1 Custom ecrecover Implementation Introduces Maintenance Risk

Status: Fixed

Description

Manual signature parsing is error-prone and less secure than OpenZeppelin's widely audited battle-tested ECDSA.recover that also includes additional safety checks (e.g., ensuring the recovered address is not address(0)) and handles edge cases more robustly.

Remediation

Use ECDSA.recover directly for safer and cleaner code.

2.4.2 Magic Numbers Reduce Code Clarity

Status: Fixed

Description

Hardcoding values like 1 and >= 2 obscures logic meaning and increases risk of errors.

Remediation

Define constants at the top of the contract and replace the numbers in the code:

```
uint256 public constant PAYMENT_TYPE_NATIVE = 1;
uint256 public constant MIN_TOKEN_PAYMENT_TYPE = 2;
```

2.4.3 Missing Events on Withdrawals

Status: Fixed

Description

Withdrawals are not tracked via events, making them hard to monitor through explorers or dApp UIs.

Remediation

Add NativeWithdrawn and TokenWithdrawn events and emit them upon successful withdrawals.

CONSCLUSION

This security audit found three high-severity findings and three low-severity findings.

Critical:

High:

- Excessive ERC-20 Permit Allowance: Users' funds are at severe risk due to buyItemToken requesting unlimited spending approvals, potentially allowing an attacker to drain all tokens from user wallets if the contract is targeted in a supply chain attack.
- Gas Limit Exceeded (DoS): Large airdrop batches (airdropBatchPayment and airdropBatchPaymentNative) can consume excessive gas, leading to transaction reverts, loss of gas fees for the owner, and a Denial-of-Service for the airdrop operation.
- Reverting Recipient Contract (DoS): A single malicious or non-compliant contract address in an airdrop batch can cause the entire airdropBatchPayment transaction to revert, resulting in failed distributions and gas loss for the owner.

Low:

- Custom ecrecover Implementation: The contract uses a custom function for signature recovery instead of a more robust, audited, and standard library like OpenZeppelin's ECDSA.recover, increasing the risk of subtle bugs.
- Magic Numbers: Hardcoded numerical values are used where named constants would improve code readability, maintainability, and reduce the likelihood of errors.
- Missing Events: Withdrawal functions (withdrawNative and withdrawToken) do not emit events, hindering off-chain tracking and transparency for monitoring and UI tools.

The Delabs team has addressed the key issues identified in the audit, including airdrop robustness, and enhancing code clarity. With these fixes in place, the contract is significantly more secure, reliable, and aligned with the best practices.s

REFERENCES

[1] https://medium.com/@JohnnyTime/solidity-smart-contract-unbounded-loops-dos-attack-vulnerability-explained-with-real-example-f4b4aca27c08