Delabs Games
Boxing Star X Wallet (TON)

Security
Audit
Report

February 3, 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 (Ton Coin) and Jetton 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	31.01.2025 - 04.02.2025
Number of Auditors	3

Project Last Log

Date	Data Hash	Note
10.11.2022	oF2+dscXTQbm3d0rVQqD7j2rhtcKojvRZqBRwqr4VPU=	Fixes

Project Scope

The audit covered the following files:

File name	Link
bsx_wallet.fc	https://verifier.ton.org/EQA5aB9c0hmtVHsaXpwGhic5doEmUID_NueRQviJnItEwYFd?testnet=
helpers.fc	https://verifier.ton.org/EQA5aB9c0hmtVHsaXpwGhic5doEmUID_NueRQviJnItEwYFd?testnet=

1.5 Summary of findings

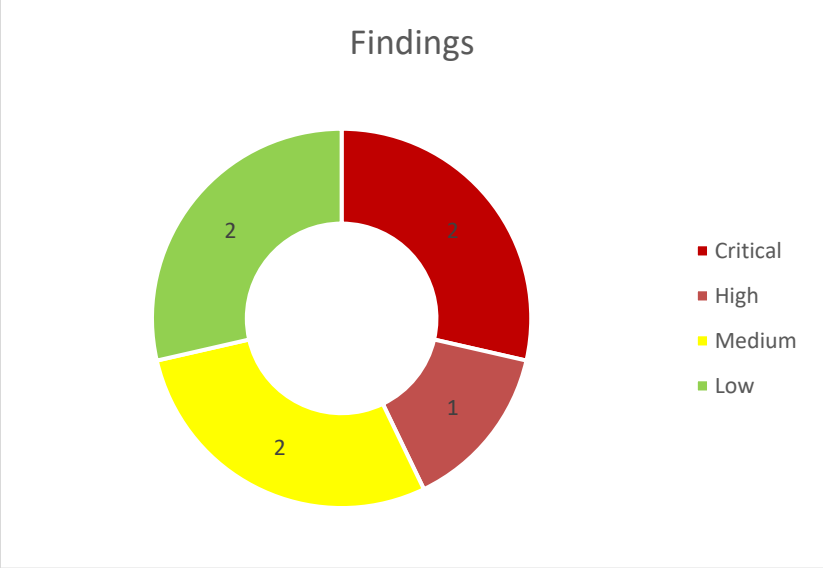

Figure 1. Findings chart

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

2. FINDINGS REPORT

2.1 Critical

2.1.1 Missing validation and emitting events in transfer_notification() function

Status: Fixed

Description

Anyone can craft an op::transfer_notification transaction and send it to bsx_wallet which will let it pass as "valid" even though the actual payment to a jetton wallet might be missing. As a result, it may confuse the off-chain backend application which could credit the game items by mistake.

Here is the vulnerable piece of code. The validation whether the transaction came from Jetton Wallet is missing.

```
    if (op == op::transfer_notification) {
        int jetton_amount = in_msg_body~load_coins();

        throw_unless(87, jetton_amount > 0);

        cell ref = in_msg_body~load_ref();
        slice ref_cs = ref.begin_parse();
        int item_id = ref_cs~load_uint(64);
        int quantity = ref_cs~load_uint(64);
        int item_price = 0;

        (slice data, int found?) = itemPrices.udict_get?(64, item_id);

        if (found?){
            item_price = data~load_coins();
            throw_unless(501, jetton_amount >= item_price * quantity);
        } else {
            throw(502);
        }

        return ();
    }

```

In addition, the transfer_notification and buy_item_ton should log the purchase events according to the latest recommendations from TON.

Remediation

To fix these issues, two things should be done.

1. Introduce a condition check that only transactions received from the corresponding jetton wallet addresses are accepted.
2. Log the event so the backend application would not have to look at multiple contracts [1].

Here is an example of code fixing the problems.

```
const int error::unauthorized_address = 401;
const int op::purchase_event = 0x59f24awa;
const int log::purchase = 0x01; ;; topic
const int send::regular = 0;
...
() emit_log(int topic, builder log) impure inline_ref {
    cell msg = begin_cell()
        .store_uint(0x31, 6)       ;; ext_out_msg_info$11 (binary 110001)
        .store_uint(256, 9)        ;; external address length = 256 bits
        .store_uint(topic, 256)    ;; put the "topic" as the external address
        .store_uint(0, 64 + 32 + 1)
        .store_builder(log)        ;; actual payload
        .end_cell();
    send_raw_message(msg, send::regular);
}
...
cell ref = in_msg_body~load_ref();
slice from_address = in_msg_body~load_msg_addr();
...
if (op == op::transfer_notification) {
    ...
    if (found?){
        item_price = data~load_coins();
        slice jetton_wallet_address = data~load_msg_addr()
        throw_unless(error::unauthorized_address, equal_slices(sender_address, jetton_wallet_ad-
dress))
    int total_price = item_price * quantity
    ... ;; checks
    builder b = begin_cell()
        .store_slice(buyer_address)
        .store_uint(item_id, 64)
        .store_uint(quantity, 64)
        .store_coins(total_paid);
    emit_log(log::purchase, b);
...
```

The same log should be added to buy_item_ton()

2.1.2 Items price error

Status: Fixed

Description

When the owner sets the price for an item, the price is defined as a single number. A buyer can buy game items for TON or Jetton. However the price is not the same for TON and Jetton. It will only be the same if t1 Jetton is equal to 1 TON Coin, which is most likely not the case. Thus, the price should be different for TON and Jetton. Consider an example: the price for an item is 10 in your smart contract. That means a user can buy it for 10 TON Coin or 10 Jetton Coins, but is 1 Jetton Coin equal to 1 TON Coin?

Remediation

Add a second field, so the dictionary would store two prices for TON and Jetton.

```
if (op == op::set_item_price) {
...
    int ton_price = in_msg_body~load_coins();       ;; the new TON price
    int jetton_price = in_msg_body~load_coins();    ;; the new Jetton price
...
    ;; Build a cell storing both prices and the wallet.
    cell message = begin_cell()
        .store_coins(ton_price)
        .store_coins(jetton_price)
        .store_slice(jetton_wallet_address)
        .end_cell();

    ;; Save it in the dictionary at key = item_id
    itemPrices~udict_set(64, item_id, message.begin_parse());
```

Functions get_item_data, transfer_notification and buy_item_ton must be adjusted accordingly.

2.2 High

2.2.1 Last-second price change (front-running)

Status: Acknowledged

Description

If the administrator is malicious (or you plan to allow other users to sell game items), the buyer can be deceived into paying much more for an item than they agreed to.

Currently, there is no measure to protect a buyer from a last-second price change by the owner (or another seller in the future). E.g., a user sends a buy_item_ton message (with a certain expected price), but the owner changes the price in the same block (or right before). The contract sees only the new price at execution time.

Remediation

If price changes are frequent, consider storing a “price lock” or “block timestamp” (i.e. allow to buy the item only after 10 blocks) to protect a buyer’s transaction from being front run by an owner’s price change. Or you can incorporate an approach where the buyer’s message includes the price the buyer expects, and you compare it to the dictionary price.

2.3 Medium

2.3.1 No refund on error or overpayment

Status: Acknowledged

Description

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

This does not protect the buyer from making a mistake.

Remediation

There are 2 options here.

1. Do nothing

2. Calculate and return the remaining value after the purchase

In this case, the contract should return the amount equal to (msg_value - total_price - GAS_FEE)

Here is what refund in *buy_item_ton()* may look like

```
const int GAS_FEE = 10000000;  ;; 0.01 TON
 ...
() send_refund(slice recipient, int payment) impure {

    raw_reserve(payment, 0);

   ;; Return the remaining value back to the sender after collecting the payment.
    cell msg = begin_cell()
            .store_msg_flags_and_address_none(NON_BOUNCEABLE)
            .store_slice(recipient)
            .store_coins(0)
            .store_prefix_only_body()
            .store_uint(op::top-up, 32)
            .store_uint(query_id, 64)
            .end_cell();
    send_raw_message(msg, 64);
}

;; If user overpaid, refund the difference.
int overpay = msg_value - total_price - GAS_FEE;
if (overpay > 0) {
    send_refund(sender_address, total_price + GAS_FEE);
}
```

2.3.2 Buying items with 0 quantity

Status: Fixed

Description

In the current implementation, the only validation check in buying functions is whether used paid enough. However, it is possible to buy items with 0 quantity with a successful transaction. Since there is no point in selling 0 items and wasting gas, such a transaction should be denied.

```
throw_unless(501, msg_value >= item_price * quantity);
```

Potential missing validation checks on the off chain backend application may lead to crediting game items to the actor for free.

Remediation

Enforce the `*quantity must be more than 0*` policy in buying functions.

```
throw_unless(501, quantity > 0);
throw_unless(501, msg_value >= item_price * quantity);
```

2.4 Low

2.4.1 Limited amount of game items for sale

Status: Fixed

Description

The storage of a smart contract in TON is limited to 65536 unique cells, so the maximum number of entries in the dictionary is 32768 (at best). The bigger your dictionary, the deeper the cell tree. The deeper the tree, the more gas it costs to read/modify.

Remediation

The current implementation is acceptable but not recommended. When you get to larger amounts of data, consider splitting the data across multiple contracts, also known as TON Sharding [2].

2.4.2 The contract does not return the gas excess

Status: Acknowledged

Description

If excess gas is not returned to the sender, the funds will accumulate in your contracts over time.

Pros of sending the remaining gas back:

Users do not have to guess the exact minimal value for each transaction. They can send a bit more for gas overhead without losing the remainder.

Remediation

```
() send_excesses(slice excesses_address) impure inline_ref {
  cell msg = begin_cell()
    .store_msg_flag(msg_flag::non_bounceable)
    .store_slice(excesses_address)
    .store_coins(0)
    .store_uint(0, 1 + 4 + 4 + 64 + 32 + 1 + 1);
    .store_op(op::excesses)
    .store_query_id(ctx::query_id)
    .end_cell();

  send_raw_message(msg, mode::carry_remaining_gas);
```

The optimal practice is to return excess gas on every user transaction.

CONSCLUSION

The Boxing Star X Wallet smart contract is straightforward but contains several significant vulnerabilities (2 Critical, 1 High, 2 Medium, 2 Low). The following issues were identified:

1. Critical:

   • Missing validation and event logging in transfer_notification(): Allows unauthorized calls that can fool off-chain systems into crediting items without actual payment.

   • Single-price model for both TON and Jetton: Implies 1:1 equivalence between TON and Jetton when setting item prices, which rarely holds true and can lead to under/overcharging.
2. High:

   • Front-Running on Prices: A malicious or inattentive owner (or future sellers) can adjust item prices at the last moment, causing buyers to pay more than expected.
3. Medium:

   • No Refund Mechanism: Overpayments or errors in transaction amounts result in the contract retaining excess funds indefinitely.

   • Purchasing 0 Quantity: The contract fails to reject buy requests with zero quantity, potentially letting a malicious user exploit off-chain logic to claim items incorrectly.
4. Low:

   • Limited Number of Game Items: TON's cell and dictionary limitations mean large data sets can become unwieldy and expensive to manage, though not immediately critical.

• No Gas Excess Return: Users providing extra gas cannot recover the remainder, which can lead to inefficiencies and higher operational costs.

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://github.com/HipoFinance/contract/blob/main/contracts/imports/utils.fc#L45-L59

[2] TON Blog on Sharding

[3] https://docs.ton.org/v3/guidelines/smart-contracts/security/secure-programming#:~:text=If%20excess%20gas%20is%20not,this%20is%20just%20suboptimal%20practice.

[4] TON Smart Contracts Security Guidelines

[5] https://github.com/HipoFinance/contract/blob/main/contracts/imports/utils.fc