Whales DMCC Holders Contracts

Security Assessment

April 4, 2025

Prepared for:
Sergei Iudin
Whales DMCC

Prepared by: **Elvis Skoždopolj and Guillermo Larregay**

Table of Contents

Project Summary

Contact Information

The following project manager was associated with this project:

Jeff Braswell, Project Manager
jeff.braswell@trailofbits.com

The following engineering director was associated with this project:

Jim Miller, Engineering Director, Blockchain & Cryptography
james.miller@trailofbits.com

The following consultants were associated with this project:

Elvis Skoždopolj, Consultant
elvis.skozdopolj@trailofbits.com

Guillermo Larregay, Consultant
guillermo.larregay@trailofbits.com

Project Timeline

The significant events and milestones of the project are listed below.

Date	Event
February 26, 2025	Pre-project kickoff call
March 10, 2025	Status update meeting #1
March 14, 2025	Delivery of report draft
March 14, 2025	Report readout meeting
April 4, 2025	Delivery of final comprehensive report
May 12, 2025	Completion of fix review

Executive Summary

Engagement Overview

Whales DMCC engaged Trail of Bits to review the security of the `jetton-card-v3` and `ton-card-v8` smart contracts in the `holders-contracts` repository. These contracts implement a payment card system on the TON blockchain, allowing controlled spending with configurable limits. The spending is controlled by a privileged actor, which is currently a centralized wallet.

A team of two consultants conducted the review from March 3 to March 14, 2025, for a total of four engineer-weeks of effort. The component highlighted in this report was reviewed in the second week of the engagement, for a total of two engineer-weeks of effort. Our testing efforts focused on review of the privileged roles, their privileges, and their limitations; analysis of potential race conditions, denial-of-service attack vectors, and gas consumption; and analysis of bounceable messages, bouncing conditions, and reversal of state. We investigated if user funds can be stolen or permanently locked, if access controls or spending limits can be bypassed, and if users can upgrade their cards to an unexpected version or a different type of card.

With full access to source code, we performed static testing of the targets, using manual processes. Because the codebase is a complex system, several components such as the back end and front end were out of scope for the audit, and their interactions with the contracts were not specified. The off-chain parts of the system along with key management practices were not reviewed. The `holders-contracts` repository contains multiple different smart contracts, but only the `jetton-card-v3` and `ton-card-v8` contracts were considered in scope. The full list of coverage limitations can be found in the Project Coverage section.

Observations and Impact

The `jetton-card-v3` and `ton-card-v8` contracts contain a lot of code duplication that could be reduced by moving the shared code into another component. We did not have access to any documentation apart from some inline code documentation; such documentation would have improved the reviewability of the code.

We discovered multiple race condition issues (TOB-WHC-2, TOB-WHC-4, TOB-WHC-5) that could have been caught by creating and comparing sequence diagrams of message flows. Finding TOB-WHC-2 was a known issue present in an earlier version of the `jetton-card-v3` contract and was later reintroduced into the version under review. This indicates that development practices could be improved, such as by keeping track of all known issues and creating unit tests for each that can be regularly run during development or in the CI process. Multiple issues are a result of insufficient constraints: TOB-WHC-3 and TOB-WHC-1 demonstrate inconsistency in applying constraints on the same or similar

operations, TOB-WHC-9 allows for nonsequential sequence numbers, and TOB-WHC-6 may allow users to update their card code to a different type of card or to an older version of the code.

The codebase is permissioned, and the controller and owner have control over user funds. The access controls are robust, and the system is not designed to be decentralized. Nevertheless, limiting the powers of privileged roles by implementing timelocks or user opt-out mechanisms could be beneficial, along with creating user-facing documentation outlining the risks related to interacting with the system. Writing a system specification detailing behavior and constraints would go a long way toward improving the codebase.

Recommendations

Based on the codebase maturity evaluation and findings identified during the security review, Trail of Bits recommends that Whales DMCC take the following steps:

- **Remediate the findings disclosed in this report.** These findings should be addressed as part of a direct remediation or any refactor that may occur when addressing other recommendations.
- **Create user- and developer-facing documentation.** The documentation that we had access to is sparse, consisting of mostly inline code comments. Proper code documentation that includes message flows, user stories, and sequence diagrams should be created in order to make further development, review, and maintenance easier.
- **Refactor the card contracts.** The jetton card and TON card contracts contain duplicate code that could be moved into a dependency.
- **Retroactively create a system specification.** Creating a system specification that includes expected system behavior and operation constraints would help with discovering and preventing the reintroduction of security issues and would help in enforcing important operation constraints.
- **Perform another security review after the findings are addressed and the code is refactored.** Complex contracts such as `jetton-card-v3` and `ton-card-v8` should be reviewed in depth and with better access to documentation and back-end processes.

Finding Severities and Categories

The following tables provide the number of findings by severity and category.

EXPOSURE ANALYSIS

Severity	*Count*
High	0
Medium	3
Low	3
Informational	5
Undetermined	0

CATEGORY BREAKDOWN

Category	*Count*
Cryptography	1
Data Validation	6
Denial of Service	2
Timing	2

Project Goals

The engagement was scoped to provide a security assessment of the Whales DMCC jetton card and TON card smart contracts. Specifically, we sought to answer the following non-exhaustive list of questions:

- Can user funds become permanently locked?
- Can user funds be stolen?
- Can a denial-of-service attack be performed on user cards?
- Can a user's card balance become larger than the jetton wallet balance?
- Can someone reduce a user's card balance?
- Can users update their cards to different types?
- Are card updates safe? Could users mistakenly break functionality or lose funds during updates?
- Can users or privileged roles bypass the withdrawal and transfer limitations?
- Can the card spending limits be bypassed?

Project Targets

The engagement involved reviewing and testing the target listed below.

holders-contracts

Repository	https://github.com/whalescorp/holders-contracts
Version	`389aaa9a0f4342e869d7439665d5d51a047ac1ee`
Type	FunC
Platform	TON

Project Coverage

This section provides an overview of the analysis coverage of the review, as determined by our high-level engagement goals. Our approaches included the following:

- Manual review of the main message flows in the `jetton-card-v3` and `ton-card-v8` contracts, including the following:
    - An investigation of race conditions to see if they can lead to incorrect balance updates, denial-of-service attacks, temporary or permanent freezing of user funds, or funds being stolen
    - An investigation of the deposit and withdrawal flows to see if users can bypass any of the withdrawal checks or inflate their balance
    - An investigation of the card closure process via the `op::close` operation and via a text command to identify if a card can be prevented from being closed, if user funds can be lost when a card is closed, if balances can be incorrectly withdrawn, and if a card can be reopened once closed
    - A review of the `op::update` operation to see if users can update the code of the contracts without the approval of the stored key, update the code to an older version, or update the code to a different card type
    - A review of the limits and how they are updated, looking for ways in which limits could be bypassed or incorrectly updated
    - A review of the roles, their privileges, and their limitations, including a manual analysis of the ability to use members' funds, and of the overall fund-handling code for any potential issues
    - Analysis of bounceable messages, bouncing conditions, and reversal of state
    - Review of message values and gas consumption
- Manual review of the testing suite to gain context on expected system behavior

Coverage Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. The following list outlines the coverage limitations of the engagement and indicates system elements that may warrant further review:

- We did not have access to specific documentation about the system. The team provided a high-level description of the system, how the contracts interact with each other, and the actual repositories.

- The front end, user interface, back end, and all other components of the system not mentioned in the coverage section were not in scope.
- We did not consider front-end checks or validations.
- The treasure contract was explicitly out of scope. It was covered only to understand the interactions between it and the card contracts and the general dynamic of asset flows. Even though we identified issues related to the treasure contract through this limited coverage, we recommend conducting a deeper review of this contract after those issues are fixed.
- All additional files in the project, such as deployment scripts, tests, wrappers, and any non-FunC files, were not part of the scope and, therefore, were not reviewed.
- The following are coverage limitations in the `holders-contracts` repository:
    - Time zone and date calculations were not reviewed in depth. External libraries and functions such as `unixToMonth` and `tsToDMY` were not considered in scope.
    - The jetton sync process in the `jetton-card-v3` contract was reviewed, but given the number of issues found, we recommend having a deeper review performed.
    - Due to the code duplication and minimal differences between similar functions in the `ton-card-v8` and `jetton-card-v3` contacts, flows that both contracts have in common should be reviewed again after refactoring and common code should be moved to a new import file.

Codebase Maturity Evaluation

Trail of Bits uses a traffic-light protocol to provide each client with a clear understanding of the areas in which its codebase is mature, immature, or underdeveloped. Deficiencies identified here often stem from root causes within the software development life cycle that should be addressed through standardization measures (e.g., the use of common libraries, functions, or frameworks) or training and awareness programs.

Category	Summary	Result
Arithmetic	No arithmetic issues were discovered during the review. The contracts use simple formulas for balance tracking, the only exception being the Unix-to-month conversion formula, which should be documented better.	**Satisfactory**
Auditing	We are not aware of any off-chain monitoring system that can detect or prevent malicious behavior. The team has not provided documentation about any incident response plans. Functions do not emit external messages or logs when critical storage state is modified.	**Further Investigation Required**
Authentication / Access Controls	The repository has clearly defined roles for all contracts, and access controls are correctly implemented for all privileged functions. Some roles can be changed or revoked; however, the public key of the account that signs the code update data is immutable, which could prevent recovery in case of a private key leak (TOB-WHC-8). Additionally, some crucial operations are missing constraints (TOB-WHC-1, TOB-WHC-6) or could be made safer by making them two-step operations (TOB-WHC-7). We are not aware of how privileged accounts are protected, especially the ones controlled by the back end. For each non-user account with privileges, the documentation should specify who has access to it, if it is a single key or a multisignature wallet, how it is protected, and what the procedure is if the key is leaked.	**Moderate**
Complexity	There is significant code duplication between both card	**Moderate**

Management	contracts that should be refactored. Some operation handlers such as `op::execute` are complex, as they perform several actions. These kinds of functions should be refactored into simpler parts, to improve code quality and readability. All complex operations should have comprehensive inline documentation.	
Cryptography and Key Management	The signature verification public key for the jetton card and TON card contracts is immutable (TOB-WHC-8), which can pose a significant risk if the private key is leaked. We did not review the key management practices of the protocol team or how the account used to sign code upgrades is guarded.	**Further Investigation Required**
Decentralization	None of the contracts in scope are meant to be decentralized. The contracts are upgradeable; however, we found issues with the upgradeability mechanisms, such as TOB-WHC-6 and TOB-WHC-7. In general, there are no timelocks or opt-out mechanisms for users to react when a system parameter is changed. It would be beneficial to clearly document the privileged role powers and how they can impact user funds; this should be made available as part of the user-facing documentation.	**Weak**
Documentation	We had no access to documentation during the audit. The code comments are insufficient and can be improved, especially for complex functions such as jetton card balance syncing. The documentation should include architecture descriptions and diagrams, message flow diagrams, and user stories. All critical components should be identified and their risks clearly described. Retroactively writing a system specification would make it easier to discover security issues and guide the improvement of the testing suites.	**Weak**
Low-Level Manipulation	The contracts in scope do not use any low-level manipulation.	**Not Applicable**

Testing and Verification	The test suites are limited in general and consider only “happy paths.” Some tests do not run out of the box. Due to time constraints, the testing suites were reviewed only to gain context on message flows and expected system behaviors.	**Further Investigation Required**
Transaction Ordering	The codebase contains multiple race conditions, such as TOB-WHC-4 and TOB-WHC-5. One race condition, TOB-WHC-2, was fixed in an earlier version of the jetton card contract but was later reintroduced in this version. This indicates that the development practices of the team could be improved by clearly tracking known issues, creating documentation, writing a specification, and including unit tests for all known security issues.	**Weak**

Summary of Findings

The table below summarizes the findings of the review, including details on type and severity.

ID	Title	Type	Severity
1	Payment card authority could potentially bypass whitelist limits for jetton transfers	Data Validation	**Informational**
2	User card balance can become permanently locked	Timing	**Medium**
3	A closed card can be reopened	Data Validation	**Medium**
4	Users can be prevented from syncing their balance	Denial of Service	**Low**
5	The execution operation is vulnerable to denial-of-service attacks	Denial of Service	**Medium**
6	User code update procedure is insufficiently constrained	Data Validation	**Low**
7	Updates to treasure code and data are irreversible	Data Validation	**Informational**
8	The public key of the signature verification scheme is immutable	Cryptography	**Informational**
9	Sequence numbers are not enforced to be sequential	Data Validation	**Low**
10	Deployment process for card contracts can be vulnerable to front-running	Timing	**Informational**
11	Time zone handling does not account for varying time zones	Data Validation	**Informational**

Detailed Findings

1. Payment card authority could potentially bypass whitelist limits for jetton transfers

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-WHC-1
Target: `holders-contracts/packages/ton/contracts/treasure-v0.fc`	

Description

The `authority` role in the treasure contract can use the `op::transfer` operation to bypass the jetton transfer destination checks performed in the `op::transfer_jettons` operation. This allows them to transfer any jetton to any address, even addresses that are not whitelisted.

The `op::transfer_jettons` operation allows the `authority` role to send the jettons held by the stored `acc_jetton_wallet` account to any whitelisted destination, if the treasure contract is configured with the `is_for_jetton?` state variable set to true. These checks are shown in figure 1.1:

```
if (op == op::transfer_jettons) {
    [...]

    ;; Allow transfers to this address from this jetton wallet if:
    ;; treasure is for TONs OR jetton wallet matched and destination is whitelisted
    var (is_for_jetton?, acc_code, _, acc_jetton_wallet) =
load_accounts_config(config);
    var is_whitelisted? = check_transfer_destination(whitelist, destination,
forward_payload);

    ;; If jetton wallet is like configured one and destination is whitelisted
    throw_unless(error::invalid_address, is_for_jetton? & equal_slice_bits(jw,
acc_jetton_wallet) & is_whitelisted?);

    throw_if(error::not_enough_gas, msg_value < config::jetton_transfer_fees +
forward_ton_amount);

    do_send_jettons(
        send_mode::carry_remaining_value,
        0,
        jw,
        query_id,
        value,
```

```
        destination,
        response_destination,
        custom_payload,
        forward_ton_amount,
        forward_payload
    );

    return ();
}
```

Figure 1.1: The `op::transfer_jettons` operation in the `treasure-v0` contract (holders-contracts/packages/ton/contracts/treasure-v0.fc#L282–L318)

However, in the `op::transfer` operation, we can see that the `authority` role is intended to provide any arbitrary payload and send a message to any destination if the `is_for_jetton?` state variable is set to true. This is shown in figure 1.2:

```
if (op == op::transfer) {
    var destination = in_msg_body~load_msg_addr();
    var value = in_msg_body~load_coins();
    var forward_payload = in_msg_body;

    ;; Allow transfers to this address if:
    ;; treasure is for jettons OR destination is whitelisted
    var (is_for_jetton?, acc_code, _, acc_jetton_wallet) =
load_accounts_config(config);
    throw_unless(error::invalid_address, (~ is_for_jetton?) &
check_transfer_destination(whitelist, destination, forward_payload));

    ;; Reserve storage
    raw_reserve(max(config::storage_reserve, my_balance - value - msg_value), 0);

    ;; Send transfer to recepient
    do_send_message_simple(destination, send_mode::separate_gas, value,
forward_payload);

    ;; Refund gas to authority
    do_send_message(sender_address, send_mode::carry_remaining_balance,
op::transfer::callback, query_id, 0, forward_payload);

    return ();
}
```

Figure 1.2: The `op::transfer` operation in the `treasure-v0` contract (holders-contracts/packages/ton/contracts/treasure-v0.fc#L256–L276)

Due to this, the `authority` role could bypass the whitelist checks performed in the `op::transfer_jettons` operation and transfer any jetton to any arbitrary address.

This issue is currently not exploitable since the implementation differs from the intended implementation stated in the code comments. However, if the access control was implemented as intended this would be considered a high-severity issue.

Exploit Scenario

The treasure contract holds $1 million worth of USDC, and it is configured to allow jetton transfers only to two KYC-verified partner institutions. However, Eve compromises the `authority` role and uses the `op::transfer` operation to transfer the entire balance to her own account.

Recommendations

Short term, add a check that ensures the `op::transfer` operation throws an error if the `destination` address is the predefined jetton wallet. Update the access control implementation to match the code comments.

Long term, carefully analyze the constraints that should be placed on message flows that involve asset transfer and improve the coverage of the testing suite. Make sure to include common adversarial scenarios.

2. User card balance can become permanently locked

Severity: **Medium**	Difficulty: **Medium**
Type: Timing	Finding ID: TOB-WHC-2
Target: `holders-contracts/packages/ton/contracts/jetton-card-v3.fc`	

Description

A user's jetton card balance can become permanently locked due to a race condition between the `op::close` operation and the `op::start_balance_sync` operation.

The user or the controller of the jetton card contract can initiate the closure of the payment card. When a card is closed, the entire leftover balance is withdrawn to the user and the treasure contract, depending on the value of the state variables that track each balance.

```
;; Perform completition
if (complete) {
    var balanceA = ctx_deposited_a + ctx_transferred_b - ctx_transferred_a -
ctx_withdrawn_a;
    var balanceB = ctx_deposited_b + ctx_transferred_a - ctx_transferred_b -
ctx_withdrawn_b;
    if (balanceA > 0) {
        ctx_withdrawn_a = ctx_withdrawn_a + balanceA;
        do_send_token_message(ctx_min_tons_for_token_fee,
ctx_min_tons_for_token_forward, ctx_address_a, sender_address, send_mode::default,
callback::withdraw, query_id, balanceA, extras);
    }
    if (balanceB > 0) {
        ctx_withdrawn_b = ctx_withdrawn_b + balanceB;
        do_send_token_message(ctx_min_tons_for_token_fee,
ctx_min_tons_for_token_forward, ctx_address_b, sender_address, send_mode::default,
callback::withdraw, query_id, balanceB, extras);
    }
}
```

Figure 2.1: A snippet of the `do_uncooperative_close` function (holders-contracts/packages/ton/contracts/jetton-card-v3.fc#L451–L463)

However, the `op::close` operation does not check if a balance sync is currently in progress. Since the `op::start_balance_sync` operation is callable by anyone, and it transfers the entire jetton balance of the contract to itself, a race condition can cause the `op::close` operation's withdrawal to silently fail. In this case, the user and treasure contract will receive no jettons and all of the jettons will become stuck in the jetton card contract's jetton wallet. The only way to remedy this is to upgrade the jetton card contract's code.

This issue was fixed in an earlier version of the jetton card contract and was later reintroduced in this version.

Exploit Scenario

Alice holds $10,000 of USDC in her jetton card and initiates the closure procedure for the card. Eve notices this and waits for the closure delay to pass. Once it has passed, she calls the `op::start_balance_sync` operation once per block. The transfer from this operation lands in the same block as the closure withdrawal but is executed first, causing the withdrawal transfer to silently fail. Alice's funds become permanently locked in the jetton card contract.

Recommendations

Short term, if a card is closed, allow the user to withdraw their jettons by using the `op::withdraw_jettons` operation. Add a check to the close operation so it throws an error if a sync is in progress.

Long term, create sequence diagrams for each message flow in the system contracts and compare the message flows that update or rely on the same contract state. This will make it easier to discover race conditions. Implement tracking of known vulnerabilities present in past versions of contracts that are in active development, and add tests for each vulnerability in order to ensure past vulnerabilities are not reintroduced in future versions.

3. A closed card can be reopened

Severity: **Medium**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-WHC-3
Target: `holders-contracts/packages/ton/contracts/jetton-card-v3.fc`, `holders-contracts/packages/ton/contracts/ton-card-v8.fc`	

Description

A jetton card or TON card that has been previously closed can be reopened by the user by sending a text command to the card.

The jetton cards and TON cards implement a closure mechanism that is intended to permanently close a user's card. This mechanism can be initiated by the user or by the controller, either by executing the `op::close` command or by sending a text command "Close" to the contract.

The `op::close` command will throw an error if the `ctx_state` is equal to `state::closed`, as shown in figure 3.1:

```
;; Preconditions
throw_unless(error::invalid_message, ctx_seed == seed); ;; Check if message seed is
correct
throw_unless(error::invalid_message, tag == tag::close); ;; Check if message tag is
correct
throw_unless(error::closed, ctx_state != state::closed); ;; Check if not closed
throw_unless(error::inconsistent_state, seqno == ctx_seqno); ;; Check if seqno is
correct
```

Figure 3.1: A snippet of the `op::close` command in the `jetton-card-v3` contract (`holders-contracts/packages/ton/contracts/jetton-card-v3.fc#L857–L861`)

However, the text command code does not perform the same validation, as shown in figure 3.2:

```
        ;; Close
        throw_unless(error::invalid_message, in_msg_body.slice_bits() >= 40);
        var text = in_msg_body~load_uint(40);
        if (text == "Close"u) { ;; Close
            throw_if(error::invalid_message, in_msg_body.slice_bits() > 0);

            do_uncooperative_close(sender_address, 0, 0, in_msg_body);
            return ();
        }
```

Figure 3.2: A snippet of the "Close"u text command in the jetton-card-v3 contract (holders-contracts/packages/ton/contracts/jetton-card-v3.fc#L1015–L1023)

The do_uncooperative_close function will update the ctx_deadline and then set the ctx_state variable to state::requested_close_a if the caller is the user and ctx_state is not already state::requested_close_a. This is shown in the highlighted lines in figure 3.3:

```
() do_uncooperative_close(slice sender_address, int index, int query_id, slice
extras) impure {
    var complete = false;
    if (ctx_state == state::requested_close_a) {
        if (index == 0) {
            throw_unless(error::inconsistent_state, now() > ctx_deadline); ;; Check
deadline passed
        }
        complete = true;
    } elseif (index == 0) {
        ctx_deadline = now() + ctx_close_timeout;
    } elseif (index == 1) {
        complete = true;
    }

    [...]

    ;; Persist
    if (complete) {
        ctx_state = state::closed;
    } elseif (index == 0) {
        ctx_state = state::requested_close_a;
    }
    store_data();
}
```

Figure 3.3: A snippet of the do_uncooperative_close function in the jetton-card-v3 contract (holders-contracts/packages/ton/contracts/jetton-card-v3.fc#L438–L475)

Since a closed card's ctx_state is state::closed, this allows us to reopen a closed card in a pending closure state. The pending closure state does not limit user actions in any way and is equivalent to the open state.

This issue is present in the jetton-card-v3 and ton-card-v8 contracts.

Exploit Scenario

Alice, the contract controller, forcibly closes Eve's card account due to suspicious activity. Eve sends a text message "Close" and reopens her account in a pending closure state and continues using it.

Recommendations

Short term, update the text command “Close” so that it throws an error if the card is already closed.

Long term, ensure that similar or the same operations that can be executed in multiple ways always validate the same data. This can be achieved by writing a detailed constraint specification for each operation in the system and by adding unit tests for each constraint. The constraints should be well documented in internal developer documentation and inline code documentation.

4. Users can be prevented from syncing their balance

Severity: **Low**	Difficulty: **Low**
Type: Denial of Service	Finding ID: TOB-WHC-4
Target: `holders-contracts/packages/ton/contracts/jetton-card-v3.fc`	

Description

A user can be indefinitely prevented from syncing their card's jetton balance.

The jetton balance of the `jetton-card-v3` contract can become out of sync if the transferred jettons did not trigger a `transfer_notification`. Due to this, a syncing operation, `op::start_balance_sync`, is implemented that is callable by anyone (figure 4.1).

```
if (op == op::start_balance_sync) {
    ;; Reserve due payment for storage fees
    raw_reserve(due_payment(), 4);

    ;; Min 0.01 + fees TON
    throw_if(error::invalid_message, msg_value < 10000000 + due_payment() +
ctx_min_tons_for_token_fee + ctx_min_tons_for_token_forward);

    var query_id = in_msg_body~load_uint(64);
    throw_unless(error::invalid_message, query_id > 0); ;; Allow only positive
query_id
    var expected_balance = in_msg_body~load_coins();
    in_msg_body.end_parse();

    var balanceA = ctx_deposited_a + ctx_transferred_b - ctx_transferred_a -
ctx_withdrawn_a;
    var balanceB = ctx_deposited_b + ctx_transferred_a - ctx_transferred_b -
ctx_withdrawn_b;

    var contractReservedBalance = balanceA + balanceB;

    throw_unless(error::insufficient_balance, contractReservedBalance <
expected_balance);

    ctx_sync_in_progress? = true;
    ctx_sync_query_id = query_id;

    do_send_token_message(
        0,
        ctx_min_tons_for_token_forward,
        my_address(),
```

```
        sender_address,
        send_mode::carry_remaining_balance,
        callback::sync_balance,
        query_id,
        expected_balance,

begin_cell().store_ref(begin_cell().store_coins(contractReservedBalance).store_slice
(sender_address).end_cell()).end_cell().begin_parse()
    );

    store_data();

    return ();
}
```

Figure 4.1: The `op::start_balance_sync` operation in the `jetton-card-v3` contract (holders-contracts/packages/ton/contracts/jetton-card-v3.fc#L894–L931)

This operation sets the `ctx_sync_in_progress?` state variable to true and the `ctx_sync_query_id` state variable to an arbitrary user-provided value. Then it transfers the jettons from the jetton wallet of the card back to the same wallet in order to trigger a `transfer_notification` and update the stored jetton balance of the contract.

A snippet of the `transfer_notification` operation is shown in figure 4.2:

```
if (ctx_sync_in_progress? & (ctx_sync_query_id == query_id) &
(contractReservedBalance == before_sync_balance)) {
    ;; Reset sync
    ctx_sync_in_progress? = false;
    ctx_sync_query_id = 0;
    sync_success = true;

    var amount = expected_balance - before_sync_balance;

    ;; deposit to user's side
    ctx_deposited_a = ctx_deposited_a + amount;

    store_data();
}

   ;; Send message anyway
    do_send_message_builder(
        refund_address,
        send_mode::carry_remaining_value + send_mode::ignore_errors,
        response::balance_sync,
        query_id,
        0,
        begin_cell()
            .store_int(sync_success, 1)
            .store_coins(ctx_deposited_a)
    );
```

```
    return ();
}

;; Reset sync if sync in progress
if (ctx_sync_in_progress?) {
    ctx_sync_in_progress? = false;
    ctx_sync_query_id = 0;
}
```

Figure 4.2: A snippet of the `transfer_notification` operation in the `jetton-card-v3` contract (`holders-contracts/packages/ton/contracts/jetton-card-v3.fc#L599–L632`)

However, we can force the check on the first highlighted line of figure 4.2 to fail by simply sending another `op::start_balance_sync` message to the `jetton-card-v3` contract with a different `query_id` value, overriding the stored `ctx_sync_query_id` value.

Additionally, we can cancel the sync by transferring a small amount of the jetton to the card, triggering the second highlighted code portion in the `transfer_notification`. This would prevent a user from syncing their jetton balance for an indefinite amount of time.

Exploit Scenario

Alice transfers $10,000 worth of USDC to her jetton card contract but mistakenly does not provide a forward TON amount, so the `transfer_notification` is not triggered. Eve notices this and starts spamming calls to the `op::start_balance_sync` operation, preventing Alice from syncing her balance. Eve requests a ransom payment in exchange for stopping the denial-of-service attack.

Recommendations

Short term, make the `op::start_balance_sync` operation permissioned. The user of the card and a privileged role determined by the protocol team should be the only ones allowed to sync the card balance.

Long term, create sequence diagrams for each message flow and compare the message flows that write to or read from the same state in order to more easily discover race conditions.

5. The execution operation is vulnerable to denial-of-service attacks	
Severity: **Medium**	Difficulty: **Low**
Type: Denial of Service	Finding ID: TOB-WHC-5
Target: `holders-contracts/packages/ton/contracts/jetton-card-v3.fc`	

Description

Calls to the `op::execute` operation can be prevented by initiating a balance sync, which can be initiated multiple times and by anyone.

The `op::execute` operation is the main balance-modifying operation and is callable only by the controller. This operation can transfer or withdraw jettons from either the user or the treasure account; however, this operation cannot be executed if a sync is currently in progress:

```
throw_if(error::sync_in_progress, ctx_sync_in_progress?); ;; Check if sync is in
progress
```

Figure 5.1: A snippet of the `op::execute` operation in the `jetton-card-v3` contract (`holders-contracts/packages/ton/contracts/jetton-card-v3.fc#L700`)

Since the `op::start_balance_sync` operation is callable by anyone and can be executed multiple times, this allows a malicious actor to indefinitely prevent user cards from functioning correctly.

Exploit Scenario

Alice deposits $10,000 worth of USDC jettons to her card and initiates a transfer in order to pay for a service. However, Eve spams messages with the `op::start_balance_sync` operation and prevents the controller from executing the transaction. This temporarily prevents Alice from being able to use her card and causes the controller to waste TON on gas.

Recommendations

Short term, make the `op::start_balance_sync` operation permissioned. The user of the card and a privileged role determined by the protocol team should be the only ones allowed to sync the card balance. Update the check on line 911 of the `jetton-card-v3` contract to throw an error if the `expected_balance` value is smaller than or equal to the current balance.

Long term, create sequence diagrams for each message flow and compare the message flows that write to or read from the same state in order to more easily discover race conditions.

6. User code update procedure is insufficiently constrained

Severity: **Low**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-WHC-6
Target: `holders-contracts/packages/ton/contracts/jetton-card-v3.fc`, `holders-contracts/packages/ton/contracts/ton-card-v8.fc`	

Description

Users are able to update the code of their jetton cards and TON cards to any version that was ever released, including older versions. Additionally, if the public key used for signing is the same for both contracts, a jetton card can be upgraded to a TON card and vice versa.

The `op::update` operation allows a user to update the code of their contract if the new contract code was hashed and signed by the stored key in `ctx_key_updates`. This operation is shown in figure 6.1:

```
if (op == op::update) {
    throw_unless(error::not_authorized, equal_slice_bits(sender_address,
ctx_address_a)); ;; throw if not user
    throw_if(error::invalid_message, msg_value < 50000000); ;; 0.05 TON

    var query_id = in_msg_body~load_uint(64);
    var code = in_msg_body~load_ref();

    ;; ;; Check signature validity
    var signature = in_msg_body~load_ref().begin_parse();
    throw_unless(error::invalid_message, signature.slice_bits() == 512);

    ;; ;; Check hash
    var hash = cell_hash(code);
    throw_unless(error::not_authorized, check_signature(hash, signature,
ctx_key_updates)); ;; verify signature

    ;; ;; Update code
    set_code(code);

    do_send_message(sender_address, send_mode::carry_remaining_value,
response::update, query_id, 0, in_msg_body);
    return ();
}
```

Figure 6.1: The `op::update` operation in the `jetton-card-v3` contract (`holders-contracts/packages/ton/contracts/jetton-card-v3.fc#L871–L891`)

However, this operation does not prevent the code from being updated to an older version of the contract. Additionally, if the `ctx_key_updates` state variable is the same in the `jetton-card-v3` and `ton-card-v8` contracts, users can update their cards to a different type of card, which will completely and permanently break the cards' functionality.

Exploit Scenario

Alice sees a new update for the `ton-card-v8` contract but mistakenly submits it to her jetton card. Since the `ctx_key_updates` public key is the same for both the `jetton-card-v3` and `ton-card-v8` contracts, the update passes but Alice's jetton card becomes permanently nonfunctional.

Recommendations

Short term, ensure that a unique key is used for the `ton-card-v8` and `jetton-card-v3` contracts. Consider disabling code version downgrade in order to prevent users from exploiting known issues of older code versions.

Long term, consider implementing a two-step update procedure that requires an external validation of the update hash, in order to avoid bricking the contracts. In this scenario, users would propose the update, a message will be sent to this external actor to verify the hash and wallet type, and if it is valid, a new message accepting the update will be sent to the card contract.

7. Updates to treasure code and data are irreversible

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-WHC-7
Target: `holders-contracts/packages/ton/contracts/treasure-v0.fc`	

Description

The `op::update` operation of the treasure contract updates the code and the data of the contract at once. If this update is incorrect, there is no way to recover from it.

```
if (op == op::update) {
    var new_code = in_msg_body~load_ref();
    var new_data = in_msg_body~load_ref();
    set_data(new_data);
    set_code(new_code);

    do_send_message(sender_address, send_mode::carry_remaining_value,
op::update::response, query_id, 0, null());
    return ();
}
```

Figure 7.1: The `op::update` operation in the `treasure-v0` contract (`holders-contracts/packages/ton/contracts/treasure-v0.fc#L401–L409`)

Recommendations

Short term, use a two-step process with a timelock to ensure that incorrect code or data updates are not immediately irreversible.

Long term, identify and document all possible actions that can be taken by privileged accounts, along with their associated risks. This will facilitate reviews of the codebase and prevent future mistakes.

8. The public key of the signature verification scheme is immutable

Severity: **Informational**	Difficulty: **High**
Type: Cryptography	Finding ID: TOB-WHC-8
Target: `holders-contracts/packages/ton/contracts/jetton-card-v3.fc`, `holders-contracts/packages/ton/contracts/ton-card-v8.fc`	

Description

The `ctx_key_updates` storage variable contains the public key used to verify the signature on code updates. There is no clear specification about how this key is generated and how the associated private key is stored or protected in the back end.

If the private key is leaked, anybody can sign arbitrary updates to card contracts and change their functionality. There is currently no way to prevent this scenario, as the public key is constant and immutable in all card contracts, and the only requirement for updating a contract's code is a message from the user.

```
if (op == op::update) {
    throw_unless(error::not_authorized, equal_slice_bits(sender_address,
ctx_address_a)); ;; throw if not user
    throw_if(error::invalid_message, msg_value < 50000000); ;; 0.05 TON

    var query_id = in_msg_body~load_uint(64);
    var code = in_msg_body~load_ref();

    ;; ;; Check signature validity
    var signature = in_msg_body~load_ref().begin_parse();
    throw_unless(error::invalid_message, signature.slice_bits() == 512);

    ;; ;; Check hash
    var hash = cell_hash(code);
    throw_unless(error::not_authorized, check_signature(hash, signature,
ctx_key_updates)); ;; verify signature

    ;; ;; Update code
    set_code(code);

    do_send_message(sender_address, send_mode::carry_remaining_value,
response::update, query_id, 0, in_msg_body);
    return ();
}
```

Figure 8.1: Signature verification using the hash of the code cell (holders-contracts/packages/ton/contracts/ton-card-v8.fc#L673–L693)

Recommendations

Short term, allow a trusted actor to update the public key used to verify signatures in case they have to be changed.

Long term, consider on-chain alternatives for signature verification, such as a variation of the verification scheme recommended in the long-term recommendations for issue TOB-WHC-6. Implement industry-standard data verification algorithms and processes, avoiding single points of failure for critical validations.

9. Sequence numbers are not enforced to be sequential

Severity: **Low**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-WHC-9
Target: `holders-contracts/packages/ton/contracts/ton-card-v8.fc`, `holders-contracts/packages/ton/contracts/jetton-card-v3.fc`	

Description

In both card contracts, the `ctx_seqno` variable is used as a nonce-like, ever-increasing sequence number to ensure that old transactions cannot be replayed once they are executed. A similar variable, `limits_seqno`, is used for the limits dictionary to ensure that there can never be more than three pending limits at a time.

However, the code never checks that the sequence numbers are increasing by one every time. Card users and the controller can set an arbitrary value as the new sequence number, and as long as it is greater than the stored value, it is accepted.

If the maximum value allowed in the 32-bit integer data type is used as sequence number, then it is no longer possible for the controller to call `op::execute` or for the user to call `op::update_limits`, since the greater-than check will always fail.

```
if (op == op::update_limits) {
    [...]
    var new_onetime = in_msg_body~load_coins();
    var new_daily = in_msg_body~load_coins();
    var new_monthly = in_msg_body~load_coins();
    var new_seqno = in_msg_body~load_uint(32);
    var extras = in_msg_body;

    [...]

    var (
        limit_onetime,
        limit_daily,
        limit_monthly,
        spent_daily,
        spent_monthly,
        daily_deadline,
        monthly_deadline,
        limits_seqno,
        pending_limits
    ) = load_limits();
```

```
    throw_if(error::invalid_message, new_seqno <= limits_seqno);
    if (~ cell_null?(pending_limits)) {
        var (max_limits_seqno, _, _) = pending_limits.udict_get_max_ref?(32);
        var (min_limits_seqno, _, _) = pending_limits.udict_get_min_ref?(32);

        throw_if(error::invalid_message, new_seqno <= max_limits_seqno);
        throw_if(error::too_many_pending_limits, (max_limits_seqno -
min_limits_seqno) >= 4);
    }

    var new_limits = build_pending_limits(new_onetime, new_daily, new_monthly,
new_seqno);
    pending_limits = pending_limits.udict_set_ref(32, new_seqno, new_limits);
    store_limits(limit_onetime, limit_daily, limit_monthly, spent_daily,
spent_monthly, daily_deadline, monthly_deadline, limits_seqno, pending_limits);
    [...]
}
```

Figure 9.1: Uses of sequence numbers in `op::update_limits` (holders-contracts/packages/ton/contracts/ton-card-v8.fc#L600–L649)

Exploit Scenario

Bob wants to update his card limits, so he sends an `op::update_limits` message. He does not remember his last sequence number, so he sets it to `4294967295`, the maximum value for a 32-bit unsigned integer.

When the operation is processed, `limits_seqno` is set to the value used in the message, and the `op::update_limits` functionality is permanently broken, as there is no valid next sequence number for updating the limits.

Recommendations

Short term, enforce the sequence numbers to increase by one.

Long term, review every assumption or invariant in the system and ensure proper data validation is performed. Improve the test suites with adversarial cases, and ensure that test coverage is high enough for all files.

10. Deployment process for card contracts can be vulnerable to front-running

Severity: **Informational**	Difficulty: **Low**
Type: Timing	Finding ID: TOB-WHC-10
Target: `holders-contracts/packages/ton/contracts/ton-card-v8.fc`, `holders-contracts/packages/ton/contracts/jetton-card-v3.fc`	

Description

The deployment and initialization process for card contracts is not necessarily atomic, as not all the required state variables are stored in the contract deployment init data. To finish the storage initialization, an `op::deploy` message is required.

If a contract's deployment is done in a different transaction than the initialization, it is possible for an attacker to initialize the contract storage with arbitrary values.

```
if (op == op::deploy) {

    ;; Parse message
    var query_id = in_msg_body~load_uint(64);
    var seed = in_msg_body~load_uint(256);
    var ds2 = in_msg_body~load_ref().begin_parse();
    slice controller = ds2~load_msg_addr();
    int updatesKey = ds2~load_uint(256);
    var ds3 = in_msg_body~load_ref().begin_parse();
    var addressA = ds3~load_msg_addr();
    var addressB = ds3~load_msg_addr();
    int tz_offset = in_msg_body~load_int(18);
    int close_timeout = in_msg_body~load_uint(32);
    var extras = in_msg_body;

    ;; timezone offset cannot be bigger than 1 day
    throw_if(error::invalid_message, (tz_offset >= (24 * 60 * 60)) | (tz_offset <=
-(24 * 60 * 60)));

    ;; Parse state
    var ds = get_data().begin_parse();
    var deployed = ds~load_int(1);
    throw_if(error::invalid_message, deployed); ;; Check if not deployed
    var deploy_seed = ds~load_uint(256);
    throw_unless(error::invalid_message, deploy_seed == seed); ;; Check if seed
matches
    var treasure_address = ds~load_msg_addr();
    throw_unless(error::invalid_message, equal_slices(treasure_address, addressB));
;; Check if treasure address matches
    var deploy_controller = ds~load_msg_addr();
```

```
    throw_unless(error::invalid_message, equal_slices(deploy_controller,
controller)); ;; Check if treasure address matches
    ds.end_parse();

    ;; Persist
    store_deployment(seed, addressA, addressB, controller, updatesKey, tz_offset,
close_timeout);

    ;; Return message
    do_send_message(sender_address, send_mode::carry_remaining_balance,
response::deploy, query_id, 0, extras);

    return ();
}
```

Figure 10.1: The `op::deploy` handler in `ton-card-v8` (holders-contracts/packages/ton/contracts/ton-card-v8.fc#L400–L437)

Recommendations

Short term, ensure that deployment and initialization are done in the same transaction in all card contract deployments.

Long term, carefully consider how non-atomic, non-privileged operations can be used to misconfigure the system. Create internal developer guidance specifying that in deployment scripts, non-atomic initialization calls should be made either atomic or permissioned.

11. Time zone handling does not account for varying time zones

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-WHC-11
Target: `holders-contracts/packages/ton/contracts/ton-card-v8.fc`, `holders-contracts/packages/ton/contracts/jetton-card-v3.fc`	

Description

Daily and monthly limit calculations make use of time zones to calculate day and month boundaries. The user's time zone is stored as a seconds offset from the UTC timestamp but cannot be changed in the contract.

Many countries observe daylight saving time, so they change time zones twice a year. Other countries may arbitrarily change their time zones for political or economic reasons. The contracts are not prepared for these particular situations. Uses of the contracts near daily and monthly limits may incorrectly fail.

Another drawback is that UTC timestamps are meant to be universal and time zone–agnostic. Adding and subtracting offsets to reference timestamps is confusing and prone to errors, as the result effectively represents a different date and time.

Recommendations

Short term, decouple the time zone calculations from the smart contracts. Keeping track of varying time zones is not trivial, and dynamically changing time zone offsets once the contracts are deployed can lead to unexpected failures.

Long term, adopt a battle-tested, industry-standard solution to deal with timestamps and time zones. Do not reimplement general solutions to well-known hard problems.

A. Vulnerability Categories

The following tables describe the vulnerability categories, severity levels, and difficulty levels used in this document.

Vulnerability Categories	
Category	**Description**
Access Controls	Insufficient authorization or assessment of rights
Auditing and Logging	Insufficient auditing of actions or logging of problems
Authentication	Improper identification of users
Configuration	Misconfigured servers, devices, or software components
Cryptography	A breach of system confidentiality or integrity
Data Exposure	Exposure of sensitive information
Data Validation	Improper reliance on the structure or values of data
Denial of Service	A system failure with an availability impact
Error Reporting	Insecure or insufficient reporting of error conditions
Patching	Use of an outdated software package or library
Session Management	Improper identification of authenticated users
Testing	Insufficient test methodology or test coverage
Timing	Race conditions or other order-of-operations flaws
Undefined Behavior	Undefined behavior triggered within the system

Severity Levels	
Severity	**Description**
Informational	The issue does not pose an immediate risk but is relevant to security best practices.
Undetermined	The extent of the risk was not determined during this engagement.
Low	The risk is small or is not one the client has indicated is important.
Medium	User information is at risk; exploitation could pose reputational, legal, or moderate financial risks.
High	The flaw could affect numerous users and have serious reputational, legal, or financial implications.

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploitation was not determined during this engagement.
Low	The flaw is well known; public tools for its exploitation exist or can be scripted.
Medium	An attacker must write an exploit or will need in-depth knowledge of the system.
High	An attacker must have privileged access to the system, may need to know complex technical details, or must discover other weaknesses to exploit this issue.

B. Code Maturity Categories

The following tables describe the code maturity categories and rating criteria used in this document.

Code Maturity Categories	
Category	**Description**
Arithmetic	The proper use of mathematical operations and semantics
Auditing	The use of event auditing and logging to support monitoring
Authentication / Access Controls	The use of robust access controls to handle identification and authorization and to ensure safe interactions with the system
Complexity Management	The presence of clear structures designed to manage system complexity, including the separation of system logic into clearly defined functions
Cryptography and Key Management	The safe use of cryptographic primitives and functions, along with the presence of robust mechanisms for key generation and distribution
Decentralization	The presence of a decentralized governance structure for mitigating insider threats and managing risks posed by contract upgrades
Documentation	The presence of comprehensive and readable codebase documentation
Low-Level Manipulation	The justified use of inline assembly and low-level calls
Testing and Verification	The presence of robust testing procedures (e.g., unit tests, integration tests, and verification methods) and sufficient test coverage
Transaction Ordering	The system's resistance to transaction-ordering attacks

Rating Criteria	
Rating	**Description**
Strong	No issues were found, and the system exceeds industry standards.
Satisfactory	Minor issues were found, but the system is compliant with best practices.
Moderate	Some issues that may affect system safety were found.
Weak	Many issues that affect system safety were found.
Missing	A required component is missing, significantly affecting system safety.
Not Applicable	The category does not apply to this review.
Not Considered	The category was not considered in this review.
Further Investigation Required	Further investigation is required to reach a meaningful conclusion.

C. Code Quality Recommendations

The following recommendations are not associated with specific vulnerabilities. However, they enhance code readability and may prevent the introduction of vulnerabilities in the future.

- The `try_commit_limits` (`jetton-card-v3` and `ton-card-v8`) and `check_transfer_destination` functions should be marked as `impure` since they can throw errors.
- The `treasure-v0` contract reserves funds only during refund or transfer operations. Funds should be reserved in most or all operations in order to ensure the contract always has enough funds to avoid being frozen.
- Contracts should be refactored to avoid duplication. For example, all functions common to both `ton-card-v8` and `jetton-card-v3` should be moved to an `include` file.
- Some checks are not needed, such as `seed` and `tag` comparisons. The values can be read from the state, if needed, and they do not add any significant protection in the process.
- Magic numbers should be avoided, as they make code less readable, and they can introduce unexpected bugs if mistyped. One example is the `transfer_notification` opcode comparison in the treasure contract.
- `include` paths should be standardized in their references to local directories. For example, in the `ton-card-v8` contract, some `include` paths reference the current directory and some do not.
- Fix the typo in `op::ownership_assinged`. This should be `const op::ownership_assigned = "op::ownership_assigned"c;`.
- Unsigned integers can never be negative. Checks such as this one will not fail and should be removed.

D. Fix Review Results

When undertaking a fix review, Trail of Bits reviews the fixes implemented for issues identified in the original report. This work involves a review of specific areas of the source code and system configuration, not comprehensive analysis of the system.

On May 12, 2025, Trail of Bits reviewed the fixes and mitigations implemented by the Whales DMCC Holders team for the issues identified in this report. We reviewed each fix to determine its effectiveness in resolving the associated issue.

In summary, of the 11 issues described in this report, Whales DMCC Holders has resolved seven issues and has not resolved the remaining four issues. For additional information, please see the Detailed Fix Review Results below.

ID	Title	Severity	Status
1	Payment card authority can bypass whitelist limits for jetton transfers	Informational	Resolved
2	User card balance can become permanently locked	Medium	Resolved
3	A closed card can be reopened	Medium	Resolved
4	Users can be prevented from syncing their balance	Low	Resolved
5	The execution operation is vulnerable to denial-of-service attacks	Medium	Resolved
6	User code update procedure is insufficiently constrained	Low	Resolved
7	Updates to treasure code and data are irreversible	Informational	Unresolved
8	The public key of the signature verification scheme is immutable	Informational	Unresolved

9	Sequence numbers are not enforced to be sequential	Low	Resolved
10	Deployment process for card contracts can be vulnerable to front-running	Informational	Unresolved
11	Time zone handling does not account for varying time zones	Informational	Unresolved

Detailed Fix Review Results

TOB-WHC-1: Payment card authority could potentially bypass whitelist limits for jetton transfers
Resolved in commit `f159ec5`. The commit updates the access control checks for the `op::transfer` and `op::transfer_jettons` functions to match the code documentation. The `op::transfer` operation throws an error if it is executed with the treasure contract's jetton wallet as the destination. However, the `authority` is able to use either operation to send an `op::close` message to a card contract, initiating the closure procedure. This is considered acceptable by the Whales DMCC Holders team.

TOB-WHC-2: User card balance can become permanently locked
Resolved in commit `8e18653`. The `op::close` operation now throws an error if a sync is currently in progress, and a sync cannot be initiated if one is already in progress or the card is already closed. The `op::withdraw_jettons` operation now allows the owner of the card to use this operation if the card is already closed.

TOB-WHC-3: A closed card can be reopened
Resolved in commit `00b8656` and commit `ed7f606`. A check was added to the `do_uncooperative_close` function in the `ton-card-v8` and `jetton-card-v3` contracts that ensures the message will throw an error if the card is already closed.

TOB-WHC-4: Users can be prevented from syncing their balance
Resolved in commit `666c452`. The `op::start_balance_sync` operation has been made permissioned; only the user of the card or the controller can execute this operation.

TOB-WHC-5: The execution operation is vulnerable to denial-of-service attacks
Resolved in commit `666c452`. The `op::start_balance_sync` operation has been made permissioned; only the user of the card or the controller can execute this operation. This prevents a third party from performing a denial-of-service attack on the execution operation.

TOB-WHC-6: User code update procedure is insufficiently constrained
Resolved in commit `f42bad2` and commit `b95cce7`. The data that is hashed and signed for card contract updates now includes a revision identifier. This identifier is enforced to be exactly one larger than the currently saved identifier; this prevents users from upgrading their `jetton-card-v3` or `ton-card-v8` contract to an earlier version.

TOB-WHC-7: Updates to treasure code and data are irreversible
Unresolved. The client provided the following context for this finding's fix status:

> *We understand the risks of this vulnerability and we decided to keep it as a feature.*

TOB-WHC-8: The public key of the signature verification scheme is immutable
Unresolved. The client provided the following context for this finding's fix status:

> *We understand the risks of this vulnerability and the updates key is stored securely enough to comply with our internal procedures.*

TOB-WHC-9: Sequence numbers are not enforced to be sequential
Resolved in commit 96e5735. Both card contracts now enforce that the new sequence number is exactly one larger than the previous sequence number.

TOB-WHC-10: Deployment process for card contracts can be vulnerable to front-running
Unresolved. The client provided the following context for this finding's fix status:

> *We understand the risks of this vulnerability and backend checks of the contract state integrity were implemented before the audit.*

TOB-WHC-11: Time zone handling does not account for varying time zones
Unresolved. The client provided the following context for this finding's fix status:

> *This issue is considered informational. Keeping timezones in sync with local changes brings more complexity in the system. Note: this was considered as a vulnerability, but it is a lack of feature (timezone change)*

E. Fix Review Status Categories

The following table describes the statuses used to indicate whether an issue has been sufficiently addressed.

Fix Status	
Status	**Description**
Undetermined	The status of the issue was not determined during this engagement.
Unresolved	The issue persists and has not been resolved.
Partially Resolved	The issue persists but has been partially resolved.
Resolved	The issue has been sufficiently resolved.

About Trail of Bits

Founded in 2012 and headquartered in New York, Trail of Bits provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code. With 100+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

We maintain an exhaustive list of publications at https://github.com/trailofbits/publications, with links to papers, presentations, public audit reports, and podcast appearances.

In recent years, Trail of Bits consultants have showcased cutting-edge research through presentations at CanSecWest, HCSS, Devcon, Empire Hacking, GrrCon, LangSec, NorthSec, the O'Reilly Security Conference, PyCon, REcon, Security BSides, and SummerCon.

We specialize in software testing and code review projects, supporting client organizations in the technology, defense, and finance industries and government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, and Zoom.

Trail of Bits also operates a center of excellence with regard to blockchain security. Notable projects include audits of Algorand, Bitcoin SV, Chainlink, Compound, Ethereum 2.0, MakerDAO, Matic, Uniswap, Web3, and Zcash.

To keep up to date with our latest news and announcements, please follow @trailofbits on X and explore our public repositories at https://github.com/trailofbits. To engage us directly, visit our "Contact" page at https://www.trailofbits.com/contact or email us at info@trailofbits.com.

Trail of Bits, Inc.
228 Park Ave S #80688
New York, NY 10003
https://www.trailofbits.com
info@trailofbits.com

Notices and Remarks

Copyright and Distribution

Test Coverage Disclaimer

Trail of Bits performed all activities associated with this project in accordance with a statement of work and an agreed-upon project plan.

Security assessment projects are time-boxed and often rely on information provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

Trail of Bits uses automated testing techniques to rapidly test software controls and security properties. These techniques augment our manual security review work, but each has its limitations. For example, a tool may not generate a random edge case that violates a property or may not fully complete its analysis during the allotted time. A project's time and resource constraints also limit their use.