TON Foundation Multisignature Wallet

Security Assessment

March 25, 2024

Prepared for:
Dr. Ilya Elias
TON Foundation

Prepared by: **Guillermo Larregay and Tarun Bansal**

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, as well as 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 Twitter 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.
497 Carroll St., Space 71, Seventh Floor
Brooklyn, NY 11215
https://www.trailofbits.com
info@trailofbits.com

Notices and Remarks

Copyright and Distribution

Test Coverage Disclaimer

All activities undertaken by Trail of Bits in association with this project were performed in accordance with a statement of work and agreed upon project plan.

Security assessment projects are time-boxed and often reliant on information that may be 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 the controls and security properties of software. 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. Their use is also limited by the time and resource constraints of a project.

Table of Contents

Project Summary

Contact Information

The following project manager was associated with this project:

Anne Marie Barry, Project Manager
annemarie.barry@trailofbits.com

The following engineering director was associated with this project:

Josselin Feist, Engineering Director, Blockchain
josselin.feist@trailofbits.com

The following consultants were associated with this project:

Guillermo Larregay, Consultant
guillermo.larregay@trailofbits.com

Tarun Bansal, Consultant
tarun.bansal@trailofbits.com

Project Timeline

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

Date	Event
February 12, 2024	Pre-project kickoff call
March 05, 2024	Status update meeting #1
March 12, 2024	Delivery of report draft
March 12, 2024	Report readout meeting
March 25, 2024	Delivery of comprehensive report

Executive Summary

Engagement Overview

TON Foundation engaged Trail of Bits to review the security of multisignature wallet contracts. The provided contracts implement a generic N-of-M multisignature wallet that allows for the creation and execution of arbitrary orders, each composed of an arbitrary number of "actions" (i.e., messages to be sent). Access control is implemented as two different roles in the system: proposers, allowed to create orders; and signers, allowed to create and sign orders. It is also possible to change the multisignature wallet parameters, such as the threshold, signers, or proposers, via special orders.

A team of two consultants conducted the review from February 26 to March 11, 2024, for a total of four engineer-weeks of effort. Our testing efforts focused on contract interactions, order validation and execution, and access controls. With full access to source code and documentation as well as the test suite, we performed static and dynamic testing of the target contracts, using automated and manual processes.

Observations and Impact

We identified a total of three high-severity, three medium-severity, and three informational issues during the audit. The high-severity findings can have a significant impact on the wallet or the order contracts, with consequences ranging from the execution of unsigned orders to complete loss of control over the multisignature wallet. All of these high-severity issues also have a high difficulty rating, which means that exploiting them requires privileged access to the system or tricking other signers into approving malicious orders.

The Code Quality appendix and the Codebase Maturity Evaluation include recommendations that are not issues but can help to improve the codebase's robustness and readability, the test suite results, and the ability to monitor transactions from outside the blockchain.

Recommendations

Based on the codebase maturity evaluation and findings identified during the security review, Trail of Bits recommends that TON Foundation take the following steps prior to the release of the multisignature wallet codebase:

- **Remediate the findings disclosed in this report.** These findings should be addressed as part of a direct remediation or as part of any refactor that may occur when addressing other recommendations.
- **Improve the user public documentation and interface to interact with the contracts:** The current project documentation consists of a high-level description of the contracts, their interactions, and a technical explanation of the order operations

and gas calculations. End users, like signers and proposers, would benefit from having a clear description of the signing and execution flows. Additionally, wallets interacting with the multisignature wallet—which were out of scope for this audit—must support a clear visualization of the order actions to prevent executing unexpected actions hiding in nested or malicious orders, such as TOB-TON-MSIG-6 and TOB-TON-MSIG-7.

- **Improve the test suite.** Cover use cases outside the expected input and output patterns. Generate malicious or adversarial cases, and ensure that the contract outputs are correct and the system integrity is maintained. Add cases that span over longer time periods to consider the situation mentioned in TOB-TON-MSIG-8.

Finding Severities and Categories

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

EXPOSURE ANALYSIS

Severity	*Count*
High	3
Medium	2
Low	0
Informational	3
Undetermined	1

CATEGORY BREAKDOWN

Category	*Count*
Access Controls	2
Data Validation	3
Undefined Behavior	4

Project Goals

The engagement was scoped to provide a security assessment of the TON Foundation Multisignature Wallet. Specifically, we sought to answer the following non-exhaustive list of questions:

- Do all assumptions about the wallet and the orders hold?
- Can orders be executed with less than threshold signers?
- Is it possible for an attacker to prevent or alter the execution of orders?
- Can the created and not-yet-executed orders be modified or altered? Can they be modified after they are signed by at least one signer?
- Can a valid signer sign an order on behalf of other signers? Can an external attacker perform the same action?
- Are expired or invalidated orders possible to be executed? Can orders be re-executed?
- Can malicious orders alter the parameters of the multisignature wallet without the signers noticing?
- Can the multisignature wallet become unusable with an update?

Project Targets

The engagement involved a review and testing of the following target.

multisig-contract-v2

Repository	https://github.com/ton-blockchain/multisig-contract-v2
Version	`1a5005cd32a27ff4e4180fa2d80400b34ac0081e`
Type	FunC / TL-B
Platform	TON Blockchain

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 provided codebase.** The multisignature wallet implements the ability for proposers or signers accounts to deploy an arbitrary number of order contracts, where each one of those orders can contain an arbitrary number of "actions" to be executed from the multisignature wallet contract. Prior to the execution of the order's actions, a threshold of signer accounts is required to approve the order. We reviewed the implementation of the access controls for the two privileged accounts lists, and the validation of the approval messages. Additionally, we reviewed the execution flow of the approved orders, considering attack vectors such as nested orders, reentrancy, signer bypass, re-execution of existing order, and denial of service. All high- and medium-severity issues found in the audit were a result of this manual review.

- **Review of the test suite.** We manually reviewed existing cases, considering the provided documentation user flows. New cases were created to test the issues found in the manual review, and to determine their impact.

- **Compliance with the documented specifications.** We reviewed the provided documentation, considering the user flows, expected guarantees, and possible invariants of the system. The informational findings related to documentation were a result of this step.

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:

- Even though we evaluated the usage of the multisignature wallet as the administrator account for a new deployment of a TON Stablecoin minter, we did not consider the implications of using the wallet as a general-purpose multisignature wallet for interacting with arbitrary contracts. Specific use cases will require ad-hoc integration reviews.

- The team mentioned that an interface is under development to show unexecuted orders' actions and parameters. We had no access to such development and it was out of the set scope for this audit. We considered the interaction of contracts without taking into account the way the messages that target the multisignature wallet or the user's wallet are generated.

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	The arithmetic in the contracts is simple and easy to understand. Most operations are additions and subtractions on integer types.	**Strong**
Auditing	There are no logging messages emitted for critical operations or parameter changes, such as signers, proposers, and threshold. Order execution and message signing are also not logged. No logging messages are emitted for critical wallet operations, such as order execution or message signing, nor for wallet parameters changes. Setting up off-chain monitoring and early response systems requires the emission of events.	**Weak**
Authentication / Access Controls	The system implements two roles: signers and proposers. The permissions for each role are well defined. However, in TOB-TON-MSIG-6 and TOB-TON-MSIG-7, we found that it is possible for a signer to sign an order on behalf of other signers, breaking one of the fundamental assumptions of the multisignature wallet.	**Weak**
Complexity Management	Functions are short, not complex, and rely on the FunC standard library to avoid code duplication. However, the code lacks documentation and comments to improve readability.	**Moderate**
Decentralization	No single actor can change the multisignature wallet parameters without a threshold of signers to agree on the change. However, it is possible for a malicious signer	**Weak**

	or proposer to create rogue orders and bypass the threshold requirement (TOB-TON-MSIG-6 and TOB-TON-MSIG-7) and execute malicious orders. There is no upgradeability for the wallet and order contracts, but the wallet's configuration parameters can be changed after it is deployed. This feature introduces issues like TOB-TON-MSIG-1 that can make signers unable to execute new orders and lose control over the contracts owned by the multisignature wallet.	
Documentation	The documentation was provided in the form of Markdown files describing the system's architecture, glossary definitions, testing procedures and cases, and a technical description of the order execution and gas calculations. There is a mismatch between the documentation and the code, as the proposers can also be updated. In general, the source code is not documented in the form of code comments or NatSpec-like documentation blocks. Some test cases are better commented than others, but in general, the test files are sparsely documented.	**Moderate**
Low-Level Manipulation	No instances of low-level data manipulation are present in the code. All the required low-level interactions with the TVM (for example, for calculating gas usage) are done using standard library functions that wrap the low-level Fift instructions. Some of the features and instructions used are not yet live on TON main network, but are expected to go live before the release of the multisignature wallet contracts.	**Moderate**
Testing and Verification	The test suite provided has unit tests for the implemented features, but the test cases focus on the expected outcomes for sane input data, and do not cover scenarios similar to most of the issues in this report. The test suite should also include adversarial inputs to the normal user flows.	**Weak**
Transaction Ordering	We did not consider MEV bots or transaction ordering risks, as the wallet itself has no implicit front-running value. Additionally, as mentioned in the coverage	**Further Investigation Required**

	limitation section, we did not consider the integration with arbitrary third-party contracts; as a result, we assessed no transaction-ordering risks in those cases.	

Summary of Findings

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

ID	Title	Type	Severity
1	The wallet cannot execute new orders if the threshold is set to 0	Data Validation	High
2	Assumptions about sequential execution of order actions may be incorrect	Undefined Behavior	Informational
3	Assumptions about signer compromise may be inaccurate	Undefined Behavior	Informational
4	A non-executed order can be marked as executed	Data Validation	Medium
5	TON balance of non-executed or expired orders is not recoverable	Data Validation	Medium
6	Malicious orders can approve existing orders on behalf of any signer	Access Controls	High
7	Malicious orders can create and execute new orders	Access Controls	High
8	No upper limit on the expiration date of the orders	Undefined Behavior	Undetermined
9	Invalid orders can be executed, as order invalidation is not permanent	Undefined Behavior	Informational

Detailed Findings

1. The wallet cannot execute new orders if the threshold is set to 0

Severity: **High**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-TON-MSIG-1
Target: `multisig-contract-v2/contracts/multisig.func`	

Description

The multisignature wallet's proposers and signers can create new orders to modify any wallet parameter. On execution, all new parameters are correctly validated except for the `threshold` value, which can be set to zero:

```
} elseif (action_op == actions::update_multisig_params) {
    threshold = action~load_index();
    signers = action~load_nonempty_dict();
    signers_num = validate_dictionary_sequence(signers);
    throw_unless(error::invalid_signers, signers_num >= 1);
    throw_unless(error::invalid_threshold, threshold <= signers_num);

    proposers = action~load_dict();
    validate_dictionary_sequence(proposers);

    action.end_parse();
}
```

Figure 1.1: The validation for threshold allows a zero value to be set (multisig-contract-v2/contracts/multisig.func#L39–L50)

Setting the `threshold` value to zero prevents the multisignature wallet from executing orders created after the change. This leads to a loss of control over the wallet since it is not possible to revert this change after it was executed. This issue is a consequence of the following check in the `try_execute` function of the order contract:

```
() try_execute(int query_id) impure inline_ref {
    if (approvals_num == threshold) {
        send_message_with_only_body(
            multisig_address,
            0,
            begin_cell()
            .store_op_and_query_id(op::execute, query_id)
            .store_order_seqno(order_seqno)
            .store_timestamp(expiration_date)
```

```
            .store_index(approvals_num)
            .store_hash(signers.cell_hash())
            .store_ref(order),
            BOUNCEABLE,
            SEND_MODE_CARRY_ALL_BALANCE | SEND_MODE_BOUNCE_ON_ACTION_FAIL
        );
        executed? = true;
    }
}
```

Figure 1.2: The equality check in `try_execute` (`multisig-contract-v2/contracts/order.func#L109–L126`)

This code checks for exact equality between the number of approvals and the set threshold value. However, every execution path that leads to the `try_execute` function increases the `approvals_num` variable beforehand, so it is not possible for it to be equal to zero.

Exploit Scenario

A proposer or signer creates a new order to change the wallet parameters, and accidentally sets the new threshold value to zero.

If this order is executed, the wallet will not be able to execute orders created after the threshold is changed.

Recommendations

Short term, validate that the new threshold value is greater than zero when the update parameters order is executed.

Long term, improve the test suite to test parameter values outside of the "happy path"; adversarial inputs for all parameters should be considered in the test cases.

2. Assumptions about sequential execution of order actions may be incorrect

Severity: **Informational**	Difficulty: **Undetermined**
Type: Undefined Behavior	Finding ID: TOB-TON-MSIG-2
Target: `multisig-contract-v2/description.md`	

Description

The order documentation states that the order actions are executed sequentially:

```
## Execution guarantees

- Order can only be executed once.
- Order actions are executed sequentially.
- Order can't be executed after it's expiration date.
- Once approval is granted by signer, it can't be revoked.
```

Figure 2.1: Order documentation (`multisig-contract-v2/description.md#L42–L47`)

However, this is not necessarily the case. What is actually guaranteed is that the messages are sent in the same sequence as they were stored in the order, but the execution order will depend on external factors such as the target shard or the block execution limits.

Additionally, the order actions can contain an `execute_internal` action at any index, including a list of messages to be sent. The order of these messages with respect to other actions of the top-level order cannot be guaranteed.

Recommendations

Short term, fix the documentation to consider the edge cases for the execution guarantees.

Long term, consider the network characteristics and how they affect the guarantees given in the documentation. Create more thorough test cases for specific edge cases to ensure compliance.

3. Assumptions about signer compromise may be inaccurate

Severity: **Informational**	Difficulty: **Undetermined**
Type: Undefined Behavior	Finding ID: TOB-TON-MSIG-3
Target: `multisig-contract-v2/README.md`	

Description

In the multisignature wallet's guarantees, it is mentioned that signer compromise does not hinder order execution if fewer than `N` signers are compromised, `N` being the value of the threshold parameter:

```
## Guarantees

- Nobody except _proposers_ and _signers_ can initiate creation of new order, nobody
except _signers_ can approve new order.
- Change of the _signers_ set invalidates all orders with old set.
- _Signer_ compromise, in particularly compromise of less than N _signers_, does not
hinder to execute orders or to propose new ones (including orders which will remove
compromised _signers_ from the signers list)
- _Proposer_ compromise does not hinder to execute orders or to propose new ones
(including orders which will remove compromised _proposer_ from the proposers list)
- Logic of multisignature wallet can not be changed after deploy
```

Figure 3.1: Guarantees for the multisignature wallet (`multisig-contract-v2/README.md#L17-L23`)

However, this assumption is not always true. For example, in a 4-of-6 multisig (that is, a wallet with six signers and a threshold of four), if three signers are compromised, it is not possible to execute new orders, as there is no possible way of getting four non-compromised approvals.

In particular, the mentioned guarantee—the multisig wallet remains operational with a compromise of less than `N` (not equal to `N`) signers—only holds when `N <= M/2`, where `M` is the total number of signers.

Recommendations

Short term, either update the documentation and add the precondition requisite, or add a new validation before setting the `threshold` parameter to ensure that the `threshold` is less than or equal to half of the total number of the signers.

Long term, consider expanding the test suite to include edge cases and test all the assumptions at boundary conditions.

4. A non-executed order can be marked as executed

Severity: **Medium**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-TON-MSIG-4
Target: `multisig-contract-v2/contracts/order.func`	

Description

When the order meets all requirements for execution, the `try_execute` function sends a message to the multisignature wallet contract with an operation code of `op::execute`. This message carries all TON balance of the order contract, and is marked as bounceable:

```
() try_execute(int query_id) impure inline_ref {
    if (approvals_num == threshold) {
        send_message_with_only_body(
            multisig_address,
            0,
            begin_cell()
            .store_op_and_query_id(op::execute, query_id)
            .store_order_seqno(order_seqno)
            .store_timestamp(expiration_date)
            .store_index(approvals_num)
            .store_hash(signers.cell_hash())
            .store_ref(order),
            BOUNCEABLE,
            SEND_MODE_CARRY_ALL_BALANCE | SEND_MODE_BOUNCE_ON_ACTION_FAIL
        );
        executed? = true;
    }
}
```

Figure 4.1: The `try_execute` function sends a bounceable message to the multisig (`multisig-contract-v2/contracts/order.func#L109–L126`)

It is possible for the multisignature wallet to bounce this message if, for example, there was a change of the wallet parameters that invalidates old orders. In this case, the message will bounce, and the order will be marked as executed.

This is misleading because the multisignature wallet did not process the order, so technically it was not executed, just bounced. As a comparison, when the order expires, the check in the `approve` function of the `order` contract will prevent execution but will not mark it as executed.

Exploit Scenario

An order is passed to change the multisignature wallet's signers list to increase the needed threshold. Later, an old order with the previous threshold value reaches the required number of signatures and is ready to be executed.

The `op::execute` message is sent from the `order` to the multisignature wallet contract, and the order is marked as executed. When the multisig wallet checks that the execution message is valid, it will throw an exception because the threshold is not met. The message will bounce to the `order` contract with the associated value, and the order will now be impossible to execute.

Recommendations

Short term, define a bounce behavior and have the code handle the bounce message accordingly. If needed, consider storing execution results separated from the executed flag to store the execution state of the order.

Long term, document what the expected behavior of the order contract should be when the execution message is bounced. If required, implement an execution-retry mechanism that allows signers to send the execution message again to the multisig if the revert reasons were fixed.

5. TON balance of non-executed or expired orders is not recoverable

Severity: **Medium**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-TON-MSIG-5
Target: `multisig-contract-v2/contracts/order.func`	

Description

Several conditions could make an order impossible to execute: if the order is invalidated due to a change of the signers list, if the threshold value changes in the multisig (see TOB-TON-MSIG-4), or if the order is expired. All of these conditions make it impossible to recover the TON balance stored in the order contract.

The minimum TON balance in an order contract is given by the sum of the storage costs that the contract will have to pay until its expiration and the execution costs of the order in the multisignature wallet up to the point where `~execute_order()` function takes over using the `accept_message()` function. However, the real balance in the order contract can be higher since the message value is forwarded to the order contract when it is created:

```
send_message_with_state_init_and_body(
    order_address,
    0,
    state_init,
    init_body,
    BOUNCEABLE,
    SEND_MODE_CARRY_ALL_REMAINING_MESSAGE_VALUE | SEND_MODE_BOUNCE_ON_ACTION_FAIL
);
```

Figure 5.1: The order creation message in the multisignature wallet contract (`multisig-contract-v2/contracts/multisig.func#L143–L150`)

Exploit Scenario

An order is passed to change the multisignature wallet's parameters, and automatically all old orders cannot be executed anymore. The balances in those orders are lost.

Recommendations

Short term, implement a mechanism to recover funds when the order is no longer executable.

Long term, model and document the different order statuses, and the transitions between them. Account for all new possible statuses, currently uncovered status and implement test cases to validate the expected behavior.

6. Malicious orders can approve existing orders on behalf of any signer

Severity: **High**	Difficulty: **High**
Type: Access Controls	Finding ID: TOB-TON-MSIG-6
Target: `multisig-contract-v2/contracts/multisig.func`	

Description

An order execution can send an `init` message to any other order contract to approve it on behalf of any signer. This can be used to execute any order maliciously by hiding the `init` message as an action in another order.

The multisignature wallet contract has a special feature that, when a new order is created by a signer with the same initialization data as an existing one, it is counted as an approval:

```
} else {
    ;; order is inited second time, if it is inited by another oracle
    ;; we count it as approval
    throw_unless(error::already_inited, in_msg_body~load_index() == threshold);
    throw_unless(error::already_inited, in_msg_body~load_nonempty_dict().cell_hash()
== signers.cell_hash());
    throw_unless(error::already_inited,in_msg_body~load_timestamp() ==
expiration_date);
    throw_unless(error::already_inited, in_msg_body~load_ref().cell_hash() ==
order.cell_hash());

    int approve_on_init? = in_msg_body~load_bool();
    throw_unless(error::already_inited, approve_on_init?);
    int signer_index = in_msg_body~load_index();
    in_msg_body.end_parse();
    (slice signer_address, int found?) = signers.udict_get?(INDEX_SIZE,
signer_index);
    throw_unless(error::unauthorized_sign, found?);
    approve(signer_index, signer_address, query_id);
    return ();
}
```

Figure 6.1: The initialization check for an existing order (`multisig-contract-v2/contracts/multisig.func#L143–L150`)

This approval is done using only the `signer_index` variable from the initialization message, looking for the `signer_address` in the `signers` dictionary.

A malicious actor could create an order with actions that approve existing orders on behalf of the other signers by sending `init` messages from the multisignature wallet to the existing order contract.

Depending on how the order actions are presented to the signers and what information they have, it is possible for them not to be aware of the exact consequences of their approval. It is also possible for the attacker to hide the malicious actions inside an `op::execute_internal` message to make the attack harder to detect if the signer interface is unclear.

Exploit Scenario

A malicious proposer creates an order to drain the multisignature wallet, but she cannot execute it. She then creates a new order that performs a seemingly innocent action, but it hides a `threshold` amount of `op::new_order` messages with the correct data for the malicious order, each with a different `signer_index` field.

The signers approve the innocent order without realizing that they will be executing an unrelated order with malicious intent. When the threshold is reached, the multisignature wallet is drained.

Recommendations

Short term, add a check in the `~execute_order` function to prevent multisignature wallet contract from sending an `init` message to any contract as an action.

Long term, document privileged roles and their permissions to ensure correct implementation of the access control checks of the system. Consider the risks of sending arbitrary messages from a privileged smart contract to arbitrary receivers and implement measures to prevent invalid authorization attacks.

7. Malicious orders can create and execute new orders

Severity: **High**	Difficulty: **High**
Type: Access Controls	Finding ID: TOB-TON-MSIG-7
Target: `multisig-contract-v2/contracts/multisig.func`	

Description

A malicious order execution can create new orders and execute them by adding the multisignature wallet as a signer and sending messages to the mutisignature wallet contract to create a new order and execute it.

Orders contain a dictionary of actions to be performed upon execution. Each of these actions can be either of type `actions::send_message`, or `actions::update_multisig_params`, and they are executed sequentially in the same order as they are stored in the dictionary. The `send_message` action allows sending an arbitrary message from the `multisig` contract to any arbitrary contract, including the `multisig` contract itself.

This allows for orders to temporarily change the wallet's parameters and execute certain actions, possibly reverting the changes before finishing the execution.

```
do {
    (action_index, slice action, int found?) =
order_body.udict_get_next?(ACTION_INDEX_SIZE, action_index);
    if (found?) {
        action = action.preload_ref().begin_parse();
        int action_op = action~load_op();
        if (action_op == actions::send_message) {
            int mode = action~load_uint(8);
            ;; Potentially multisig can init order in this request, but since
            ;; order will reject second initialisation there is no security issue
            send_raw_message(action~load_ref(), mode);
            action.end_parse();
        } elseif (action_op == actions::update_multisig_params) {
            threshold = action~load_index();
            signers = action~load_nonempty_dict();
            signers_num = validate_dictionary_sequence(signers);
            throw_unless(error::invalid_signers, signers_num >= 1);
            throw_unless(error::invalid_threshold, threshold <= signers_num);

            proposers = action~load_dict();
            validate_dictionary_sequence(proposers);
```

```
            action.end_parse();
        }
    }
} until (~ found?);
```

Figure 7.1: Update of the multisignature wallet's parameters (multisig-contract-v2/contracts/multisig.func#L28–L52)

Exploit Scenario

A malicious proposer creates an order with the following sequence of actions:

1. An `update_multisig_params` action to change the multisignature parameters, adding the multisignature wallet itself as a signer, and setting the threshold to 1.
2. A `send_message` action to create a new order with a malicious payload. This order will be executed upon creation, as the multisignature wallet is now a signer, and the threshold is 1.
3. The actions of the order created by action 2 above include an action to revert the parameters change to avoid invalidating other existing orders and to cover their tracks.

Additionally, this allows signers to execute orders that were invalidated due to a change in the signers dictionary of the multisignature wallet, provided they are not expired.

Recommendations

Short term, consider one of the following:

- Add a check in the `~execute_order` function to prevent the multisignature wallet contract from sending a message to itself if the `op` is not `execute_internal`.
- Add a check in the `~execute_order` function to ensure that the multisignature wallet contract cannot be added to the signers or proposers list.

Long term, document privileged roles and responsibilities to implement correct access control checks in the system. Consider the risks of sending arbitrary messages from a privileged smart contract to arbitrary receivers and implement measures to prevent invalid authorization attacks.

8. No upper limit on the expiration date of the orders

Severity: **Undetermined**	Difficulty: **High**
Type: Undefined Behavior	Finding ID: TOB-TON-MSIG-8
Target: `multisig-contract-v2/contracts/multisig.func`	

Description

The lack of an upper bound check on the `expiration_date` of an order can lead to the reuse of an `order_seqno` and the order contract. This can result in unexpected behavior from off-chain components relying on `order_seqno` and the order contract address.

The `new_order` operation handler code of the `multisig` contract checks if the `expiration_date` is a future date:

```
if (op == op::new_order) {
    int order_seqno = in_msg_body~load_order_seqno();
    if (~ allow_arbitrary_order_seqno) {
        if (order_seqno == MAX_ORDER_SEQNO) {
            order_seqno = next_order_seqno;
        } else {
            throw_unless(error::invalid_new_order, (order_seqno ==
next_order_seqno));
        }
        next_order_seqno += 1;
    }

    int signer? = in_msg_body~load_bool();
    int index = in_msg_body~load_index();
    int expiration_date = in_msg_body~load_timestamp();
    cell order_body = in_msg_body~load_ref();
    in_msg_body.end_parse();
    (slice expected_address, int found?) = (signer? ? signers :
proposers).udict_get?(INDEX_SIZE, index);
    throw_unless(error::unauthorized_new_order, found?);
    throw_unless(error::unauthorized_new_order, equal_slices_bits(sender_address,
expected_address));
    throw_unless(error::expired, expiration_date >= now());

    int minimal_value = calculate_order_processing_cost(order_body, signers,
expiration_date - now());
    throw_unless(error::not_enough_ton, msg_value >= minimal_value);

    cell state_init = calculate_order_state_init(my_address(), order_seqno);
    slice order_address = calculate_address_by_state_init(BASECHAIN, state_init);
    builder init_body = begin_cell()
```

```
    .store_op_and_query_id(op::init, query_id)
    .store_index(threshold)
    .store_nonempty_dict(signers)
    .store_timestamp(expiration_date)
    .store_ref(order_body)
    .store_bool(signer?);
    if (signer?) {
        init_body = init_body.store_index(index);
    }
    send_message_with_state_init_and_body(
        order_address,
        0,
        state_init,
        init_body,
        BOUNCEABLE,
        SEND_MODE_CARRY_ALL_REMAINING_MESSAGE_VALUE |
SEND_MODE_BOUNCE_ON_ACTION_FAIL
    );

} elseif (op == op::execute) {
```

Figure 8.1: The `new_order` operation handler (`multisig-contract-v2/contracts/multisig.func#L107–L152`)

However, it does not check if the `expiration_date` is not too far in the future. This allows proposers and signers to create orders with an `expiration_date` of 4–5 years in the future. This, combined with the storage rent mechanism of the TON network, can lead to a situation that allows the signers to reuse an `order_seqno` and an order contract, which can result in unexpected behavior from UI, dashboard, and other off-chain components relying on `order_seqno` and the order contract address.

Exploit Scenario

The multisig contract is configured to allow the use of arbitrary order sequence numbers. A proposer creates an order with an `expiration_date` of three years in the future. The order is approved by all the signers and executed within a month of being proposed. At the time of execution, the entire balance of the `order` contract is sent to the multisig contract. After a year, the balance of the `order` contract becomes -1 TON because of the storage fee accrual. The `order` contract is frozen, and after a couple of months, the order contract is destroyed. Later, anyone can re-deploy the `order` contract in its initial state, which is the multisig contract address and `order_seqno`. This allows the signers to create a new order with an already used `order_seqno`, approve it, and execute it after a year.

Recommendations

Short term, add a check in the `new_order` operation handler code to ensure that the `expiration_date` cannot be too far in the future.

Long term, take into consideration the network characteristics and how they affect the guarantees given in the documentation.

9. Invalid orders can be executed, as order invalidation is not permanent

Severity: **Informational**	Difficulty: **Undetermined**
Type: Undefined Behavior	Finding ID: TOB-TON-MSIG-9
Target: `multisig-contract-v2/contracts/multisig.func`	

Description

Existing orders are considered *invalid* when the multisignature wallet's parameters are changed. This, for example, prevents the execution of orders whose set of signers is different from the current set of signers defined in the wallet contract, or when the threshold was increased.

```
## Guarantees

- Nobody except _proposers_ and _signers_ can initiate creation of new order, nobody
except _signers_ can approve new order.
- Change of the _signers_ set invalidates all orders with old set.
- _Signer_ compromise, in particularly compromise of less than N _signers_, does not
hinder to execute orders or to propose new ones (including orders which will remove
compromised _signers_ from the signers list)
- _Proposer_ compromise does not hinder to execute orders or to propose new ones
(including orders which will remove compromised _proposer_ from the proposers list)
- Logic of multisignature wallet can not be changed after deploy
```

Figure 9.1: Guarantees for the multisignature wallet (`multisig-contract-v2/README.md#L17–L23`)

However, given that the signers could agree to change the multisignature wallet's parameters at any point in time, it is possible that old invalid actions become valid again, and could be executed. This can be confusing for some users if they are not aware of the implications of the parameter changes.

Recommendations

Short term, rewrite documentation to clarify that the order invalidation is not permanent.

Long term, define and document if invalidation is meant to be permanent, and modify the multisignature wallet and orders to store validity information.

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
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 is not applicable 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

This appendix contains findings that do not have immediate or obvious security implications. However, they may facilitate exploit chains targeting other vulnerabilities, become easily exploitable in future releases, or decrease code readability. We recommend fixing the issues reported here.

- **Replace magic integers in code with named constants.** In the order and wallet operation handlers, a magic constant is specified in the special case where the message operation code is zero. A new named constant should replace this to improve readability.
- **Add `impure` modifier to the `recv_internal` function in the `multisig.func` and `order.func` contracts.**
- **Fix the following typos:**
  - `op::singers_outdated` and its usage
  - `get_fee_cofigs()` (this function is not used in the codebase)

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 a comprehensive analysis of the system.

On March 29, 2024, Trail of Bits reviewed the fixes and mitigations implemented by the TON Foundation 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 nine issues described in this report, TON Foundation has resolved six issues, and has not resolved the remaining three issues. For additional information, please see the Detailed Fix Review Results below.

ID	Title	Status
1	The wallet cannot execute new orders if the threshold is set to 0	Resolved
2	Assumptions about sequential execution of order actions may be incorrect	Resolved
3	Assumptions about signer compromise may be inaccurate	Resolved
4	A non-executed order can be marked as executed	Resolved
5	TON balance of non-executed or expired orders is not recoverable	Resolved
6	Malicious orders can approve existing orders on behalf of any signer	Unresolved
7	Malicious orders can create and execute new orders	Unresolved
8	No upper limit on the expiration date of the orders	Unresolved
9	Invalid orders can be executed, as order invalidation is not permanent	Resolved

Detailed Fix Review Results

TOB-TON-MSIG-1: The wallet cannot execute new orders if the threshold is set to 0
Resolved in commit fd34ee5. The `update_multisig_params` action now validates that the new threshold value is greater than zero. An additional check was added to the multisig data getter to verify that the wallet status is correct, and the test suite was improved with two new test cases.

TOB-TON-MSIG-2: Assumptions about sequential execution of order actions may be incorrect
Resolved in commit fd34ee5. The documentation has been updated to highlight the possibility of non-sequential execution of actions of an order.

TOB-TON-MSIG-3: Assumptions about signer compromise may be inaccurate
Resolved in commit fd34ee5. The documentation now specifies that the signer compromise guarantees hold only when the number of compromised signers is less than the number of total signers minus the threshold.

TOB-TON-MSIG-4: A non-executed order can be marked as executed
Resolved in commit fd34ee5. The TON Foundation team changed the name of the `executed?` storage variable to `sent_for_execution?`. The new variable name does not cause confusion about the intended usage of the flag, and does not make any assumptions about the actual execution status.

TOB-TON-MSIG-5: TON balance of non-executed or expired orders is not recoverable
Resolved in commit fd34ee5. The expiration date is not checked in the approve function of the order, allowing any signer to try to execute an expired order. The expiration check is still done in the multisig wallet, but this change allows any signer to recover the balance stuck in an expired order. No new tests were added for this new behavior.

TOB-TON-MSIG-6: Malicious orders can approve existing orders on behalf of any signer
Unresolved. Changes implemented in commit fd34ee5. A summary of the TON Foundation statement about these changes follows:

> *We would like to point out that the contents of the order are publicly stored on the blockchain and can be read and parsed by any means.*
> *To execute any multisig orders it is required to collect a given number of approvals from signers. This is also true for the examples that are described in these issues - to execute the initial Order, it has to collect approvals by all the rules.*
> *We have added a warning in README about the need to fully familiarize yourself with the contents of an order before approving it, as well as guidelines for future multisig tools and user interfaces.*

We ask to lower Severity for these two issues in the final report to "Informational", because of course if a proposer creates something wrong, then all signers sign something wrong, then multisig will execute something wrong.
Adding this extra check produced a rather large code increase, which can be seen in the extra_checks branch, and will also prevent sending op::init from multisig to valid recipients. Adding a recipient address check would require parsing the state_init, which would increase the code even more. Given the above, we would not want to do this extra check.

We consider this to be unresolved because, as we stated, these issues should be solved at the contract level instead of relying on external user interfaces or users' awareness about what they are signing. The current status of the system makes the user vulnerable to phishing attacks, for example, trying to make them sign arbitrary messages or interact with malicious versions of the UI.

TOB-TON-MSIG-7: Malicious orders can create and execute new orders
Unresolved. See comments for the previous issue.

TOB-TON-MSIG-8: No upper limit on the expiration date of the orders
Unresolved. Commit f4c2b6f introduces changes in the documentation specifying the issues associated with `order_id` reuse.

TOB-TON-MSIG-9: Invalid orders can be executed, as order invalidation is not permanent
Resolved in commit fd34ee5. The documentation now specifies that the order is only valid when current signers are equal to signers at the time the order was created.

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.