Zellic

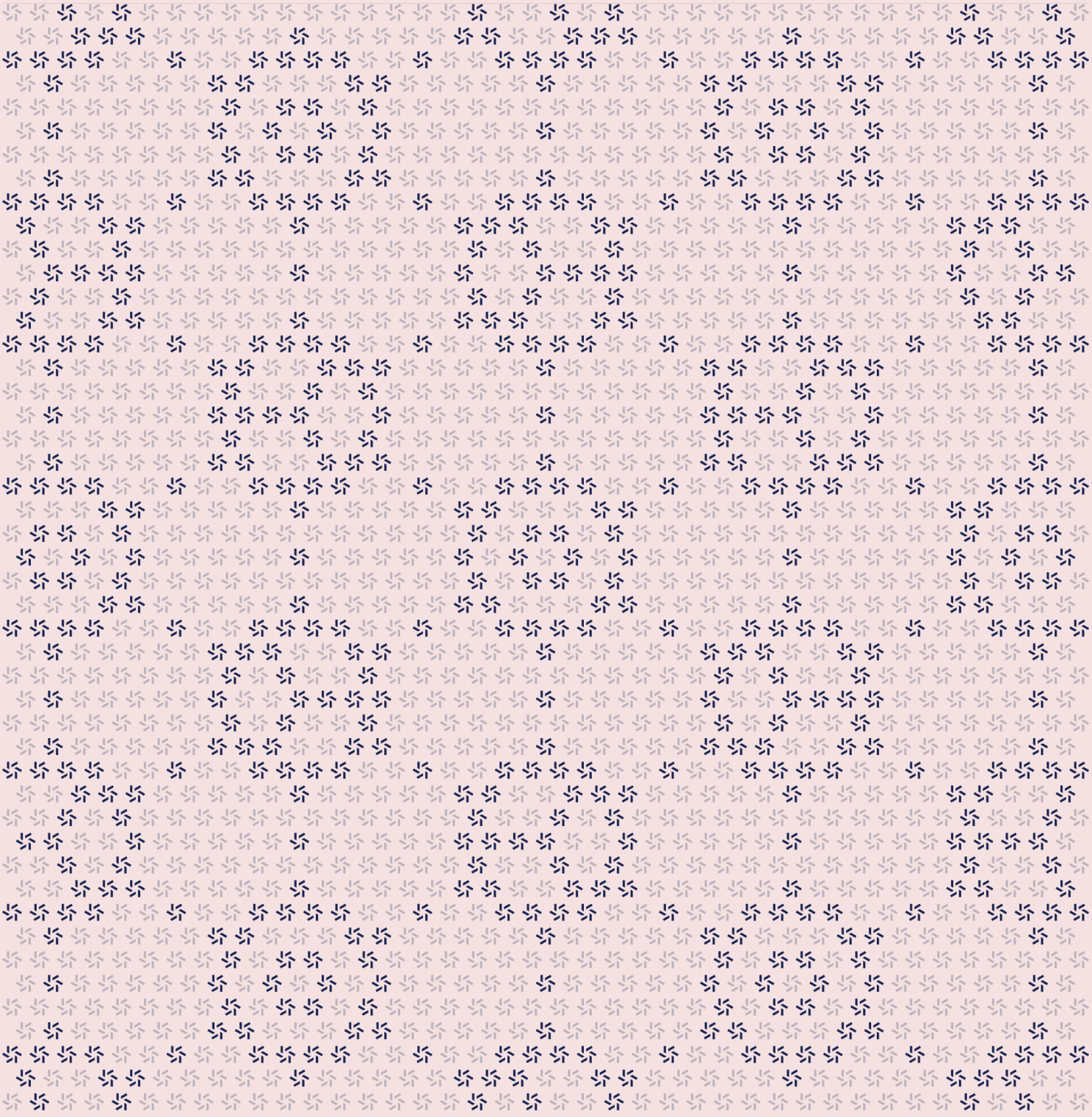
Prepared for
Dr. Elias (ilya@ton.org)
TON Foundation

Prepared by
Filippo Cremonese
Daniel Lu
Zellic

March 11, 2024

Multisig

Smart Contract Security Assessment



Contents

About Zellic **4**

1. Overview **4**

1.1. Executive Summary 5

1.2. Goals of the Assessment 5

1.3. Non-goals and Limitations 5

1.4. Results 5

2. Introduction **6**

2.1. About Multisig 7

2.2. Methodology 7

2.3. Scope 9

2.4. Project Overview 9

2.5. Project Timeline 10

3. Detailed Findings **10**

3.1. Orders are not notified of signer updates 11

3.2. Notification not sent if executing on init 13

4. Threat Model **14**

4.1. Order operations 15

4.2. Multisig operations 17

5. Assessment Results **18**

5.1. Disclaimer 19

6. Appendix A **19**

6.1. `op::execute` outbound messages are now non-bounceable 20

6.2. Additional safety checks on the threshold parameter for `actions::update_multisig_params` 20

6.3. Gas adjustments 20

About Zellic

Zellic is a vulnerability research firm with deep expertise in blockchain security. We specialize in EVM, Move (Aptos and Sui), and Solana, as well as Cairo, NEAR, and Cosmos. We review L1s and L2s, cross-chain protocols, wallets and applied cryptography, zero-knowledge circuits, web applications, and more.

Prior to Zellic, we founded the #1 CTF (competitive hacking) team ↗ worldwide in 2020, 2021, and 2023. Our engineers bring a rich set of skills and backgrounds, including cryptography, web security, mobile security, low-level exploitation, and finance. Our background in traditional infosec and competitive hacking has enabled us to consistently discover hidden vulnerabilities and develop novel security research, earning us the reputation as the go-to security firm for teams whose rate of innovation outpaces the existing security landscape.

For more on Zellic's ongoing security research initiatives, check out our website zellic.io ↗ and follow @zellic_io ↗ on Twitter. If you are interested in partnering with Zellic, contact us at hello@zellic.io ↗.

1. Overview

1.1. Executive Summary

Zellic conducted a security assessment for TON Foundation from February 14th to March 5th, 2024. During this engagement, Zellic reviewed Multisig's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.2. Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- Is the multisig authorization and authentication implemented correctly?
- Are multisig orders only executable with enough approvals from authorized addresses?
- Can orders be executed only once?
- Can orders be created only by an authorized approver or proposer?
- Do changes to the multisig configuration take effect for already existing orders?
- Are there possible DOS vectors that can be triggered by an unpriviledged attacker?

1.3. Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

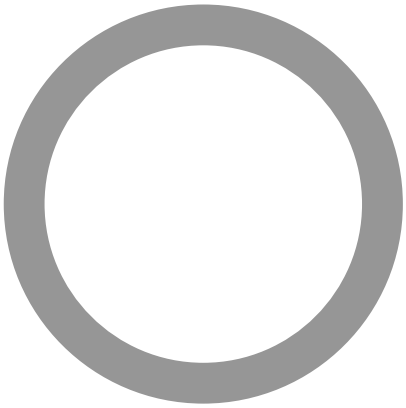
- Front-end components
- Infrastructure relating to the project
- Key custody

1.4. Results

During our assessment on the scoped Multisig contracts, we discovered two findings, both of which were informational in nature.

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	0
Medium	0
Low	0
Informational	2



2. Introduction

2.1. About Multisig

Multisig is a smart contract implementing a simple threshold-based multisig.

2.2. Methodology

During a security assessment, Zellic works through standard phases of security auditing, including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

> **Basic coding mistakes.** Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the contracts.
>
> **Business logic errors.** Business logic is the heart of any smart contract application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.
>
> **Integration risks.** Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. Time permitting, we review external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.
>
> **Code maturity.** We look for potential improvements in the codebase in general. We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradability weaknesses, centralization risks, and so on.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational"

finding higher than a “Low” finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients’ threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

2.3. Scope

The engagement involved a review of the following targets:

Multisig Contracts

Repository	https://github.com/ton-blockchain/multisig-contract-v2 ↗
Version	multisig-contract-v2: `1a5005cd32a27ff4e4180fa2d80400b34ac0081e`
Programs	• multisig.func • order.func
Type	FunC
Platform	TON

At the time this report was produced, the latest commit on the master branch was `a92d3673535eba3816bb492730bf4b9c212fa7cb`. The contracts in the most recent commit only contain minor differences with respect to the audited commit, which do not add new risks or vulnerabilities. The differences are documented in Appendix A on page 20.

2.4. Project Overview

Zellic was contracted to perform a security assessment with two consultants for a total of three engineer-weeks.

Contact Information

The following project manager was associated with the engagement:

Chad McDonald
Engagement Manager
chad@zellic.io

The following consultants were engaged to conduct the assessment:

Filippo Cremonese
Engineer
fcremo@zellic.io

Daniel Lu
Engineer
daniel@zellic.io

2.5. Project Timeline

The key dates of the engagement are detailed below. Please note that the review of the Stablecoin contract was conducted concurrently with the stablecoin-contract, which is discussed in a separate report.

February 14, 2024	Start of primary review period
March 5, 2024	End of primary review period

3. Detailed Findings

3.1. Orders are not notified of signer updates

Target	order.func		
Category	Business Logic	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The multisig contract includes functionality for updating thresholds and signers. In particular, the `update_multisig_params` order action is implemented as follows.

```
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);
```

By design, proposed orders are independent contracts that manage accounting of approvals themselves. When an order believes it can be executed, it sends a message to the multisig, where it's checked that the approvals are truly sufficient. The handler for the `execute` operation in `multisig.func` performs this check.

```
int order_seqno = in_msg_body~load_order_seqno();
int expiration_date = in_msg_body~load_timestamp();
int approvals_num = in_msg_body~load_index();
int signers_hash = in_msg_body~load_hash();
cell order_body = in_msg_body~load_ref();
in_msg_body.end_parse();

cell state_init = calculate_order_state_init(my_address(), order_seqno);
slice order_address = calculate_address_by_state_init(BASECHAIN, state_init);

throw_unless(error::unauthorized_execute, equal_slices_bits(sender_address,
    order_address));
throw_unless(error::singers_outdated, (signers_hash == signers.cell_hash()) &
    (approvals_num >= threshold));
throw_unless(error::expired, expiration_date >= now());
```

```
(threshold, signers, signers_num, proposers)~execute_order(order_body);
```

Impact

These guards handle situations where the multisig parameters are changed between order creation and execution. However, because the order contracts are not notified of updates to these parameters, it can cause some confusing behavior.

For example, if the threshold is raised, then approvals on the order can still accumulate and reach the lower threshold. When the order contract tries to initiate execution, it will get blocked by the multisig contract. The behavior when the signers are changed is similar.

If the threshold is lowered, the order will need to accumulate a number of approvals matching the threshold at creation time of the order. In other words, lowering a threshold does not apply to existing orders.

Recommendations

This behavior is due to a fundamental design choice where each order is a separate contract. It does not seem possible to efficiently propagate configuration changes to all existing orders with the current design.

Remediation

This issue has been acknowledged by TON Foundation.

3.2. Notification not sent if executing on init

Target	order.func		
Category	Business Logic	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

We observed a potentially unwanted inconsistency in the handler for `op::init`.

```
init#9c73fba2 query_id:uint64
              threshold:uint8
              signers:^(Hashmap 8 MsgAddressInt)
              expiration_date:uint48
              order:^Order
              approve_on_init:(## 1)
              signer_index:approve_on_init?uint8 = InternalMsgBody;
```

If `approve_on_init` is set and the order is not already initialized, the handler adds an approval for the signer corresponding to the given `signer_index` and invokes `try_execute`, which will execute the order if the threshold is reached (implying the threshold required a single approval).

```
if (op == op::init) {
    throw_unless(error::unauthorized_init, equal_slices_bits(sender_address,
    multisig_address));

    if (null?(threshold)) {
        ;; [...]

        int approve_on_init? = in_msg_body~load_bool();
        if (approve_on_init?) {
            int signer_index = in_msg_body~load_index();
            add_approval(signer_index);
            try_execute(query_id);
        }
        in_msg_body.end_parse();
        save_data();
        return ();
    } else {
        ;; [...]
    }
```

```
    ;; [...]
}
```

Unlike all other code paths that add an approval, this code path does not generate an outgoing message with `op::approve_accepted`.

Impact

Other contracts integrating with the multisig could rely on `op::approve_accepted` messages, therefore the lack of this outgoing message if `op::init` is used could be unexpected. The specific consequences depend on the logic of the third party contract and cannot be determined.

Recommendations

Consider generating an outgoing `op::approve_accepted` message to be consistent with all other code paths that register an approval.

Remediation

This issue has been acknowledged by TON Foundation.

4. Threat Model

This provides a description of the various operations supported by the audited contracts. As time permitted, we analyzed each operation in the contracts and created a written threat model for some critical functionality. A threat model documents a given function's externally controllable inputs and how an attacker could leverage each input to cause harm.

Not all functions in the audit scope may have been modeled. The absence of a threat model in this section does not necessarily suggest that a function is safe.

4.1. Order operations

`op::init`

```
init#9c73fba2 query_id:uint64
              threshold:uint8
              signers:^(Hashmap 8 MsgAddressInt)
              expiration_date:uint48
              order:^Order
              approve_on_init:(## 1)
              signer_index:approve_on_init?uint8 = InternalMsgBody;
```

This operation can only be invoked by the multisig associated with the order. It is used to complete initialization of the order contract.

Contract not initialized

If the contract is not already initialized, this message sets the threshold, signers set, expiration date and the message that will be sent when the order is executed. It also initializes internal state variables (`executed?`, `approvals_mask`, `approvals_num`).

If `approve_on_init` is set, an approval for the signer corresponding to the provided `signer_index` is added; this can potentially trigger execution of the order, if the required threshold is a single approval.

We note that the `signer_index` is not validated. We do not consider this to be a vulnerability, since the multisig contract validates the index and is the only allowed message sender for this operation.

Contract already initialized

If the contract is already initialized, the incoming message can be used to add an approval to the order.

The incoming message is checked to ensure it contains a threshold, signers set, expiration date and message that match the ones stored at the moment the contract was initialized (throwing an error if a mismatch is found). It also must have the `approve_on_init` flag set.

If all the above conditions hold, an approval associated to the signer corresponding to `signer_index` is added, by using the common `approve` function. The added approval can also trigger execution of the order, if the threshold is reached.

`op::approve`

```
approve#a762230f query_id:uint64
                 signer_index:uint8 = InternalMsgBody;
```

This operation can be invoked by addresses belonging to the signers set. The caller has to provide in `signer_index` the index at which the message sender address has to be found within the signers set (if not found, an error is thrown as the caller is not an authorized signer).

After checking that the sender is a signer, the common `approve` function is called to approve and potentially execute the order.

Common function `approve`

First, `approve` ensures that the order is not expired and not already executed. It then adds an approval associated to the `signer_index`. If an approval for `signer_index` already exists, an error is thrown.

If the approval is correctly registered, an `op::approve_accepted` message is sent back to the message sender. Otherwise, the error thrown in the previous step is caught and an `op::approve_rejected` message is sent to the message sender instead.

Finally, if the number of approvals has reached the required threshold, the operation is executed by sending an `op::execute` message back to the main multisig address. The message contains information that the multisig uses to validate the order, in case the threshold and/or the signer set changed after the order was created. It also carries all the remaining order contract TON balance.

Alternative approve path (`op == 0`)

The contract also supports approving an order via a message with operation value zero. The message body is required to contain the text “approve”. The text can appear directly after the zero `op` value; alternatively, it can also be formed by recursively concatenating the contents of the first referenced cell with the message body remaining after the zero `op` value.

Contrary to `op::approve`, this code path does not require to provide the index at which the message sender can be found inside the signers set; instead, it searches all the entries in the signers set until a match is found (throwing an error if the sender is not a signer).

After ensuring that the caller belongs to the signers set, the common `approve` function is used to add an approval and possibly execute the order. The logic is explained more in detail in the section documenting `op::approve`.

Bounce handling

Bounced messages are always ignored.

4.2. Multisig operations

`op::new_order`

```
new_order#f718510f query_id:uint64
                   order_seqno:uint256
                   signer:(## 1)
                   index:uint8
                   expiration_date:uint48
                   order:^Order = InternalMsgBody;
```

This operation can be used to propose a new order.

Unless the contract is configured to accept arbitrary order sequence numbers, the provided `order_seqno` is checked to ensure it matches the expected sequence number read from the contract storage. If the special value `MAX_ORDER_SEQNO` is used, the sequence number will be automatically be replaced with the correct one by the smart contract.

The multisig contract allows any of the signers or any of the members of the (possibly empty) set of proposers to propose a new order. The `signer?` field specifies whether the caller is a proposer or a signer. The address of the sender is checked to ensure it matches the address recorded at `index` in the set of signers or proposers.

The order has an expiration date, which is checked both when creating an order, and will also be checked when the order is executed. Therefore, pre-expired orders cannot be created.

The amount of TON included with the message is checked to ensure it is sufficient to cover the order processing cost.

A new contract representing the order is finally initialized and created.

`op::execute`

```
execute#75097f5d query_id:uint64
                 order_seqno:uint256
                 expiration_date:uint48
                 approvals_num:uint8
                 signers_hash:bits256
                 order:^Order = InternalMsgBody;
```

This operation can only be invoked by order contracts.

It is called when an order has received enough approvals to reach the threshold. Before executing the order, the following checks are performed:

- the caller is verified to be an order contract associated to the address of the multisig itself
- the signers set is validated to ensure it matches the current one
- the number of approvals is validated to ensure it is greater than or equal to the current threshold
- the expiry is checked

If all checks pass, `execute_order` is invoked to execute all the actions contained in the order; two types of actions are supported: `send_message` and `update_multisig_params`.

`op::send_message`

This action allows to send a completely arbitrary message. All fields of the message are controlled by the order contract invoking the multisig, including the send mode.

`op::update_multisig_params`

This action can be used to update one or more parameters of the multisig wallet: threshold, signers set, proposers set.

The signers and proposers sets are stored in dictionaries which have positive integers as keys; the dictionaries are validated to ensure the keys are non negative and contiguous (0, 1, 2, ...).

The signers set is additionally validated to ensure it contains at least one element, and at least as many elements as the threshold.

`op::execute_internal`

This operation can only be invoked by the self address; it is meant to be used by orders via `op::execute`, to chain together multiple actions.

Top up (`op == 0`)

Messages with zero operation code are simply accepted without further processing.

Bounce handling

Bounced messages are simply ignored.

5. Assessment Results

At the time of our assessment, the reviewed code was not in use on TON Mainnet.

During our assessment on the scoped Multisig contracts, we discovered two findings, both of which were informational in nature. TON Foundation acknowledged all findings and implemented fixes.

5.1. Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the version reviewed during our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.

6. Appendix A

This section discusses relevant differences between the audited commit, 1a5005cd ↗, and the most recent commit on the master branch at the time of the generation of this report, a92d3673 ↗.

The discussion is limited to the changes to the on-chain code, obtained with \texttt{git diff 1a5005cd a92d3673 -- "contracts/"}.

6.1. `op::execute` outbound messages are now non-bounceable

The `try_execute` function now sends `op::execute` messages as non-bounceable. Previous to this change, if an order execution failed the order contract received a bounced message that was immediately discarded. After this change, the order contract is oblivious to execution failures.

While such a change has the possibility to have a severe impact, in this specific case the change is inconsequential, because the order contract was already ignoring any bounced message.

6.2. Additional safety checks on the threshold parameter for `actions::update_multisig_params`

The handler for `actions::update_multisig_params` was modified to require that the `threshold` is greater than zero. While not a vulnerability on its own, and even potentially desirable in some very limited cases, allowing to set a zero threshold is antithetical to the purpose of a multisig wallet. This change prevents users from changing the threshold of a wallet to zero.

Additional sanity checks were added to `get_multisig_data` as well. The function now throws unless the wallet configuration indicates at least one signer, and a valid threshold which must be greater than zero and at most as high as the number of signers.

6.3. Gas adjustments

The gas amounts specified in `order_helpers.func` were slightly adjusted. We did not verify the new amounts to be correct, but consider the change low-risk.