Zellic

Momentum Safe

Smart Contract Security Assessment

October 28, 2022

Prepared for:

Jacky Wang

Momentum Safe

Prepared by:

Aaron Jobé and Oliver Murray

Zellic Inc.

Contents

About Zellic

Zellic was founded in 2020 by a team of blockchain specialists with more than a decade of combined industry experience. We are leading experts in smart contracts and Web3 development, cryptography, web security, and reverse engineering. Before Zellic, we founded perfect blue, the top competitive hacking team in the world. Since then, our team has won countless cybersecurity contests and blockchain security events.

Zellic aims to treat clients on a case-by-case basis and to consider their individual, unique concerns and business needs. Our goal is to see the long-term success of our partners rather than simply provide a list of present security issues. Similarly, we strive to adapt to our partners' timelines and to be as available as possible. To keep up with our latest endeavors and research, check out our website zellic.io or follow @zellic_io on Twitter. If you are interested in partnering with Zellic, please email us at hello@zellic.io or contact us on Telegram at https://t.me/zellic_io.

1 Executive Summary

Zellic conducted an audit for Momentum Safe from September 19th to September 23rd, 2022.

Our general overview of the code is that it was very well-organized and structured. The code coverage is high, and tests are included for the majority of the functions. The documentation was thorough and clearly communicated the implementation. The code was easy to comprehend, and in the majority cases, intuitive.

We applaud Momentum Safe for their attention to detail and diligence in maintaining incredibly high code quality standards in the development of Momentum Safe.

Zellic thoroughly reviewed the Momentum Safe codebase to find protocol-breaking bugs as defined by the documentation and to find any technical issues outlined in the Methodology section (2.2) of this document.

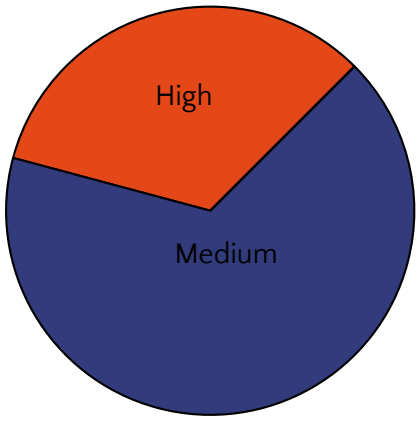
Specifically, taking into account Momentum Safe's threat model, we focused heavily on issues that would break core invariants such as calculating the positions in the pool as well as the states that the 'LendingStorageManager' handles for the liquidity providers.

During our assessment on the scoped Momentum Safe contracts, we discovered three findings. Fortunately, no critical issues were found. Of the three findings, one was High impact and two were Medium impact.

Additionally, Zellic summarized its notes and observations from the audit for Momentum Safe's benefit in the Discussion section (5) at the end of the document.

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	1
Medium	2
Low	0
Informational	0



2 Introduction

2.1 About Momentum Safe

Momentum Safe is building multi-signature products on Aptos in the style of Gnosis Safe on Ethereum. Their first product, the Momentum Safe MVP, is the core multi-signature module of protocol. It provides valuable and fundamental multi-signature technologies to the Aptos ecosystem, which will allow groups of users to decentrally manage Aptos resources and support the secure management of other protocols (e.g., DeFi).

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 open-source 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. We analyze the scoped smart contract code using automated tools to quickly sieve out and catch these shallow bugs. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so forth 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 manually review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents. We also thoroughly examine the specifications and designs themselves for inconsistencies, flaws, and vulnerabilities. This involves use-cases that open the opportunity for abuse, such as flawed tokenomics or share pricing, arbitrage opportunities, and so forth.

Complex integration risks. Several high-profile 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. We perform a meticulous review of all of the contract's possible external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich

attacks, and so forth.

Code maturity. We review for possible 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, upgradeability weaknesses, centralization risks, and so forth.

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; we assign it on a case-by-case basis based on our professional judgment and experience. As one would expect, both the severity and likelihood of an issue affect its impact; for instance, a highly severe issue's impact may be attenuated by a very low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Similarly, Zellic organizes its reports such that the most important findings come first in the document rather than being 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 numerous soft factors, such as our clients' threat models, their business needs, their project timelines, and so forth. We aim to provide useful and actionable advice to our partners that consider their long-term goals rather than simply provide a list of security issues at present.

2.3 Scope

The engagement involved a review of the following targets:

Momentum Safe Contracts

Repository	https://github.com/Momentum-Safe/momentum-safe-mvp
Versions	`3d061f6f44f1dda9f2ad7f9d62643a778fb9b916`
Programs	• registry • creator • momentum_safe • transaction • utils • test_suite
Type	Move-Aptos
Platform	Aptos

2.4 Project Overview

Zellic was contracted to perform a security assessment with two consultants for a total of two person-weeks. The assessment was conducted over the course of one calendar weeks.

Contact Information

The following project managers were associated with the engagement:

Jasraj Bedi, Co-founder
jazzy@zellic.io

Stephen Tong, Co-founder
stephen@zellic.io

The following consultants were engaged to conduct the assessment:

Aaron Jobé, Engineer
aaronj@zellic.io

Oliver Murray, Engineer
oliver@zellic.io

2.5 Project Timeline

The key dates of the engagement are detailed below.

September 19, 2022	Start of primary review period
September 23, 2022	End of primary review period

3 Detailed Findings

3.1 Wallet creation is vulnerable to front-running attacks

- **Target**: creator
- **Category**: Business Logic
- **Likelihood**: Medium
- **Severity**: High
- **Impact**: High

Description

The deployment of momentum safes is deterministic. This means that calls to `init_wallet_creation(…)` can be front-run and safe deployment can be blocked.

The call to `init_wallet_creation(…)` passes control to `init_wallet_creation_internal(…)`,

```
public(friend) fun init_wallet_creation_internal(
        s: &signer,
        owners: vector<address>,
        threshold: u8,
        init_balance: u64,
        payload: vector<u8>,
        signature: vector<u8>,
        module_address: address,
    ) acquires PendingMultiSigCreations, MultiSigCreationEvent {
…
        let public_keys = get_public_keys(&owners);
…
        let pending = borrow_global_mut<PendingMultiSigCreations>(THIS);
        let (msafe_address, nonce) = derive_new_multisig_auth_key(
            pending, signer::address_of(s), public_keys, threshold
        );
…
        // Create the momentum safe wallet and send the initial fund for
    gas fee.
        aptos_account::create_account(msafe_address);
…
    }
```

which calls `aptos_account::create_account(msafe_address);` on the deterministic ad-

dress generated from the call to `derive_new_multisig_auth_key(pending, signer::address_of(s), public_keys, threshold)`.

We created two unit tests to demonstrate this issue. For one, `test_frontrun_no` calls `init_wallet_creation` normally and passes if the call does not abort. However, `test_frontrun` calculates `msafe_address` and registers an account at that address, passing if the call to `init_wallet_creation` aborts. An excerpt of the PoC is below:

```
function test_frontrun() {
...
    let msafe_address =
    utils::address_from_bytes(utils::derive_multisig_auth_key(pubkeys,
    threshold, 0));
    // This causes the call to creator::init_wallet_creation_internal to
    fail
    aptos_account::create_account(msafe_address);

    creator::init_wallet_creation(
        owner0,
        owner_addresses,
        threshold,
        init_balance,
        test_data.wallet_creation_payload,
        init_creation_sig,
    );
...
}
```

We have provided the full PoC to Momentum Safe for reproduction and verification.

Impact

A malicious user can monitor the mempool for pending `init_wallet_creations(…)` transactions and block them by submitting transactions with a higher gas price that call `aptos_account::create_account(msafe_address)`. This is possible because the address `msafe_address` is directly readable from the mempool.

An attacker could target specific users or groups of users, or eventually take on the entire protocol.

Recommendations

Alter the design of the msafe to use nondeterministic addresses. Alternatively, if the address already exists ensure that it is a multisignature account corresponding with the set of owners and multisignature threshold of the wallet being created.

Remediation

In commit a5517d01 Momentum Safe has implemented the following fix:

```
// Create the momentum safe wallet and send the initial fund for gas fee.
if (!account::exists_at(msafe_address)) {
    aptos_account::create_account(msafe_address);
};
assert!(account::get_sequence_number(msafe_address) == 0,
    ESEQUENCE_NUMBER_MUST_ZERO);
```

If there is no aptos account at the `msafe_address` a new aptos account will be created. However, more importantly for the case of front running, if an aptos account has already been deployed than the call to `init_wallet_creation` will not fail.

This is a suitable fix because the `msafe_address` has been generated based on the rules of Aptos native multisignature framework. This ensures that only the true owners of the multisignature, as they are passed in as function arguments to `init_wallet_creation`, are in control of the `msafe_address`.

3.2 Momentum safe deployment is vulnerable to `max_gas` attacks

- **Target**: creator
- **Category**: Business Logic
- **Likelihood**: Medium
- **Severity**: High
- **Impact**: Medium

Description

When momentum safes are deployed using `momentum_safe::register(…)`, the momentum safe metadata is retrieved using a call to `creator::get_creation(…)`, as shown below:

```
public entry fun register(
        msafe: &signer,
        metadata: vector<u8>
    ) {
…
        let (owners, public_keys, nonce, threshold) =
    creator::get_creation(msafe_address);
        create_momentum(msafe, owners, public_keys, nonce, threshold,
    metadata);
…
    }
```

The call to `get_creation(…)` leads to an internal call to `simple_map::borrow(&pending.creations, &msafe_address);`:

```
public fun get_creation(
        msafe_address: address
    ): (
        vector<address>,
        vector<vector<u8>>,
        u64,
        u8
    ) acquires PendingMultiSigCreations {
…
        let creation = simple_map::borrow(&pending.creations,
    &msafe_address);
…
```

```
    }
```

The underlying `pending` data structure can be stuffed with pending safes by any user who 1) calls `registry::register(…)` and 2) repeatedly calls `creator::init_wallet_creation` with unique owners and threshold:

```
public(friend) fun init_wallet_creation_internal(
        s: &signer,
        owners: vector<address>,
        threshold: u8,
        init_balance: u64,
        payload: vector<u8>,
        signature: vector<u8>,
        module_address: address,
    ) acquires PendingMultiSigCreations, MultiSigCreationEvent {
…
        let (msafe_address, nonce) = derive_new_multisig_auth_key(
            pending, signer::address_of(s), public_keys, threshold
        );
…
        simple_map::add(&mut pending.creations, msafe_address,
    new_creation);
…
    }
```

This creates an opportunity for `max_gas` attacks because `simple_map::borrow(…);` uses a binary search algorithm, which is O(sqrt(N)).

Recommendations

Use a hash map for storing pending safe creations in the `PendingMultiSigCreations` struct.

Remediation

Momentum Safe has addressed the griefing attack vector by replacing `aptos::simple_map` with `aptos::table` in commit 18c8bbf5.

3.3 Transactions can be blocked from `max_gas` attacks

- **Target**: momentum_safe
- **Category**: Business Logic
- **Likelihood**: Medium
- **Severity**: High
- **Impact**: Medium

Description

Before transactions can be submitted for execution, all momentum safe owners must make calls to `momentum_safe::submit_signature(…)`. This retrieves the pending transaction information through a call to a `simple_map`:

```
public entry fun submit_signature(
        msafe_address: address,
        pk_index: u64,
        tx_hash: vector<u8>,
        signature: vector<u8>
    ) acquires Momentum, MomentumSafeEvent {
…
        let tx = simple_map::borrow_mut(&mut momentum.txn_book.pendings,
    &tx_hash);
…
    }
```

The underlying `pendings` data structure can be stuffed with pending safes by a malicious member of the momentum safe who repeatedly calls `momentum_safe::init_transaction(…)`:

```
public entry fun init_transaction(
        msafe_address: address,
        pk_index: u64,
        payload: vector<u8>,
        signature: vector<u8>,
    ) acquires Momentum, MomentumSafeEvent {
…
      // Validate the transaction payload
        let (tx_sn, cur_sn) = validate_txn_payload(msafe_address,
    payload);
…
        add_to_txn_book(&mut momentum.txn_book, tx_sn, new_tx);
```

```
        // Prune previous transactions with stale sequence number
        try_prune_pre_txs(&mut momentum.txn_book, cur_sn - 1);
…
    }
```

This creates an opportunity for `max_gas` attacks because `simple_map::borrow(…);` uses a binary search algorithm, which is O(sqrt(N)).

Impact

An attacker could stuff the `txn_book.pendings` to the point where the compute costs of `simple_map::borrow(…)` exceed `max_gas`. This would prevent anyone in the momentum safe from being able to sign pending transactions.

Because gas is cheap on Move-Aptos, this attack could potentially be financially feasible to a wide range of users.

Recommendations

Use a hash map for storing pending safe creations in the `PendingMultiSigCreations` struct.

Remediation

Similar to the previous finding, Momentum Safe has addressed the griefing attack vector by replacing `aptos::simple_map` with `aptos::table` in commit 18c8bbf5.

We applaud Momentum Safe for their vigilence during the auditing process. They also uncovered a similar griefing attack vector affecting the `registry::OwnerMomentumSafes` data structure. The use of `std::vector` for `OwnerMomentumSafes.pendings` and `OwnerMomentumSafes.msafes` has been replaced with a custom `table_map` located in `table_map.move`.

4 Formal Verification

The Move prover allows for formal specifications to be written on Move code, which can provide guarantees on function behavior.

During the audit period, we provided Momentum Safe with Move prover specifications, a form of formal verification. We found the prover to be highly effective at evaluating the entirety of certain functions' behavior and recommend the Momentum Safe team to add more specifications to their code base.

One of the issues we encountered was that the prover does not support bitwise operations yet.

The following is a sample of the specifications provided.

4.1 msafe::creator

Ensures the `PendingMultisigCreations` resource is created upon initialization.

```
spec init_module {
    ensures
    exists<PendingMultiSigCreations>(signer::address_of(creator));
}
```

4.2 msafe::registry

Ensures that the `OwnerMomentumSafes` resource is created upon `register`.

```
spec register {
    ensures exists<OwnerMomentumSafes>(signer::address_of(s));
}
```

4.3 msafe::transactions

Ensures that the buffer does not overflow.

```
spec set_pos_negative {
    ensures r.offset ≤ len(r.buffer);
}

spec set_pos {
    ensures r.offset ≤ len(r.buffer);
}

spec skip {
    ensures r.offset ≤ len(r.buffer);
}
```

4.4 msafe::momentum_safe

Ensures the `Momentum` resource is created upon initialization.

```
spec create_momentum {
    ensures exists<Momentum>(signer::address_of(msafe));
}
```

5 Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment.

5.1 The framework `MultiEd25519` in Momentum Safe

Momentum safe functions as a ledger that maintains storage of transaction `payloads` for groups of owners. Owners submit signed versions of the transaction `payloads`. When the number of signatures exceeds the threshold of the mutlisig any owner is able to retrieve the signatures and submit a multisig transaction under the `MultiEd25519` framework.

This means that the wallet itself does not actually execute the multi-sig transactions. It is therefore important to take a closer look at the implications of this for the design of the protocol.

Multiple momentum safes for identical groups of owners under `MultiEd25519` framework

Momentum safe leverages Aptos's native multi-signature `MultiEd25519` framework. The standard usage of `MultieEd25519` is to define the multi-signature transaction based on the set of public keys corresponding with all of the owners and the multi-sig threshold, as shown in the below snippet from Aptos documentation (https://aptos.dev/guides/creating-a-signed-transaction/#multisignature-transactions):

```
interface MultiEd25519PublicKey {
  // A list of public keys
  public_keys: Uint8Array[];
  // At least `threshold` signatures must be valid
  threshold: Uint8;
}
```

However, if the same group of owners wants to create another safe, they will be unable to do so. To get around this, momentum safe includes a dummy owner into the list of `public_keys`. This dummy owner corresponds with a nonce that counts the number of safes created by the first owner. Adding this dummy owner does not impact threshold logic because the number of signatures required to pass the transaction does not depend on the total number of owners. Furthermore, and more critically, the

private key of the nonce can never be acquired as a premise of cryptography, so this does not open up the opportunity for a non-registered owner of signing the multi-sig transaction.

During the course of the audit Momentum Safe and the Aptos Foundation discovered that the dummy owner represented by a nonce needs to correspond with a valid public key. This has been addressed in commit b649a930.

It is important to note that while the count of msafes deployed by the first owner is key in determining the address of the momentum safe, they do not have any special privileges.

Implications for testing

The address of the momentum safe address for a given set of users is known at the time of safe creation. The address is tied with the combination of owner public keys, the first owner's nonce, and the threshold. When transactions are executed for the multi-sig, they are done so from this address. ß There is a challenge in verifying the execution of multi-signature transactions from this address because it is currently not possible to write test cases for this.

5.2 Transaction ordering, cancelling, and pruning

Momentum safe uses a combination of internal business logic and protocol-level logic to enforce transaction ordering and effectively implement transaction cancelling.

Transaction ordering

Transaction ordering is achieved by leveraging the Aptos-native account-level transaction sequence number. In short, every time an account has a transaction executed, its transaction sequence number is incremented. In order for a transaction to be executed from the mempool, the transaction number in the payload must be consistent with the current transaction sequence number of the account.

New transactions can be added to the transaction book associated with a given momentum safe; however, if they are not intended to cancel or compete with previously submitted transactions, they must increment the transaction sequence number by one. This means that they can never be executed until the sufficient transactions have been executed from the corresponding momentum safe address to bring its current sequence number in alignment with the transaction sequence number.

The framework does create the potential for an interesting scenario. For example, if there are many transactions in the queue, a transaction that is indexed for later exe-

cution can be fully signed off and waiting for execution in the mempool. As the queue of transactions is processed, the preceding transaction will eventually be executed and then the pending transaction would also be executed. It could create a surprising scenario where the submission of one transaction effectively triggers another. It is hard to see how this poses a security risk, but is nonetheless important for protocol developers and users to be aware of this.

Transaction cancelling

Transaction cancelling is achieved by leveraging the framework thus described. It is possible for multiple transactions with the same transaction sequence number to be executed. Owners can then choose which one of these transactions they want to sign. The protocol-level logic that ensures alignment between the momentum safe account's current transaction sequence number and the transaction sequence number of the pending transaction means that executing one of these transactions effectively blocks any others with the same sequence number.

The suggested use case for cancelling from the in-line code documentation is to submit a competing transaction with an empty payload.

Transaction pruning

Every time a new transaction is initiated, an internal call is made to helper functions, which prune old transactions. Old transactions are identified by comparing the transaction's sequence number from the payload with the current sequence number of the momentum safe associated with the given transaction initialization.

It could also be beneficial for storage size to add a process to remove transactions that have expired based on timestamp as well.

5.3 Public key rotations

Momentum safe takes important steps to control for public key rotations.

First, they verify that the public key passed to `registry::register(…)` is consistent with the authorization key of the account of the signer. This validates that the caller is indeed passing the correct public key.

Second, they verify that the group of owner addresses passed to `init_wallet_creation` have authorization keys that are consistent with the public keys passed by each owner when they called `registry::register(…)` in the previous step.

These steps are taken to ensure that the momentum safe is deployed with groups of owners who have control over their accounts.

5.4 Test code coverage

Overall the coverage of both integration tests and unit tests is good.

However, we suggest that Test_suite.move be expanded to include integration tests demonstrating transaction ordering and transaction cancelling.

The following table summarizes the state of the unit test cases for the core modules:

Missing tests

creator.move: `remove_wallet_creation`, `clean_expired_creation`, `get_public_keys`
registry.move: `register_msafe`, `is_registered`, `add_momentum_safe`, `add_pending_safe`, `contain_address`, `find_address`
momentum_safe.move: `register`, `add_to_registry`, `prune_trx_at`

Missing negative tests

creator.move: `validate_register_payload`, `init_module`
registry.move: `get_public_key_verified`, `get_msafes_by_owner`
transaction.move: `get_module_name`, `get_function_name`
momentum_safe.move: `add_to_txn_book`
utils.move: `derive_auth_key`, `verify_signature`, `nonce_to_public_key`

Ideally, test coverage of unit tests and integration tests should be considered separately. This helps avoid the challenge of trying to reconcile coverage across complex integration tests, which may make calls to internal functions or functions on other modules. For example, `registry::register` is called in `test_suite.move`, but there is no negative test case there. Similar statements can be made for other functions missing test cases in `registry`, `creator`, and `momentum_safe`.

creator.move

The unit tests are not representative of the state of the system in regular use because there is only one owner. The positive tests for multiple owners are covered in the integration tests, but the negative tests are not. We suggest including multiple owners in the unit tests.

Include a negative test for `add_sigature` with a non-zero public key index.

transaction.move

None of the getters have test cases. In general, they are elementary; however, we suggest writing test cases for `get_module_name` and `get_function_name`.

6 Audit Results

At the time of our audit, the code was not deployed to mainnet Aptos.

During our audit, we discovered three findings, one of which was of High impact and the rest being of Medium impact.

6.1 Disclaimers

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 additional code added to the assessed project after the audit version of 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 in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but it may not be tested or functional code.

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.