Zellic

Prepared for
Larry Lyu
Left Curve Software Limited

Prepared by
Nipun Gupta
Avraham Weinstock
Zellic

April 1, 2025

Dango Account and Auth

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 Dango Account and Auth 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. Denial of service for multi-sig accounts 11

3.2. No signature is required in `RegisterUser` 13

3.3. First nonce is required to be zero 14

4. System Design **14**

4.1. Component: factory 15

4.2. Component: multi 16

4.3. Component: spot 18

4.4. Component: auth 19

5. Assessment Results 20

5.1. Disclaimer 21

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 information security 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 Left Curve Software Limited from March 21st to March 28th, 2025. During this engagement, Zellic reviewed Dango Account and Auth'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 it ensured that an unauthorized user cannot send transactions on someone else's behalf?
- Is the nonce verification working as expected?
- Are the key/parameter updates working as expected (i.e., there are no unauthorized access to these calls)?

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
- The `margin` account type and functionality specific to it not shared with other account types, such as liquidation

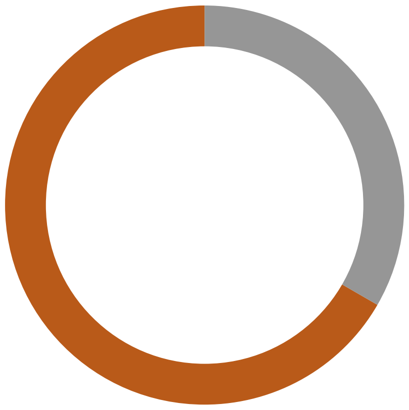
Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

1.4. Results

During our assessment on the scoped Dango Account and Auth crates, we discovered three findings. No critical issues were found. Two findings were of high impact and the remaining finding was informational in nature.

Breakdown of Finding Impacts

Impact Level	Count
🟥 Critical	0
🟧 High	2
🟨 Medium	0
🟩 Low	0
⬜ Informational	1



2. Introduction

2.1. About Dango Account and Auth

Left Curve Software Limited contributed the following description of Dango Account and Auth:

> Dango is an upcoming decentralized exchange, a novel limit order book, margin system, and user experience. The scope of this audit is Dango's account and authentication system.

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 crates.

Nondeterminism. Nondeterminism is a leading class of security issues on Cosmos. It can lead to consensus failure and blockchain halts. This includes but is not limited to vectors like wall-clock times, map iteration, and other sources of undefined behavior (UB) in Go.

Arithmetic issues. This includes but is not limited to integer overflows and underflows, floating-point associativity issues, loss of precision, and unfavorable integer rounding.

Complex integration risks. Several high-profile exploits have been the result of unintended consequences when interacting with the broader ecosystem, such as via IBC (Inter-Blockchain Communication Protocol). Zellic will review the project's potential external interactions and summarize the associated risks. If applicable, we will also examine any IBC interactions against the ICS Specification Standard to look for inconsistencies, flaws, and vulnerabilities.

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:

Dango Account and Auth Crates

Type	Rust
Platform	Cosmos

Target	left-curve
Repository	https://github.com/left-curve/left-curve ↗
Version	`17d6b0e71c2a990887a1d612933e7fbf606e7254`

Programs	`account/{factory,multi,spot}/`
	`auth/`
	`types/src/{account,account_factory}/`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of 1.8 person-weeks. The assessment was conducted by two consultants over the course of six calendar days.

Contact Information

The following project managers were associated with the engagement:

Jacob Goreski
Engagement Manager
jacob@zellic.io ↗

Chad McDonald
Engagement Manager
chad@zellic.io ↗

The following consultants were engaged to conduct the assessment:

Nipun Gupta
Engineer
nipun@zellic.io ↗

Avraham Weinstock
Engineer
avi@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

March 21, 2025	Kick-off call
March 21, 2025	Start of primary review period
March 28, 2025	End of primary review period

3. Detailed Findings

3.1. Denial of service for multi-sig accounts

Target	auth		
Category	Business Logic	**Severity**	Critical
Likelihood	Low	**Impact**	High

Description

When a user submits a transaction, it contains a `uint32` nonce in the metadata. This nonce is used to prevent replay attacks, and the function `verify_nonce_and_signature` makes sure that a nonce is not already used. The function does that by making sure that the nonce provided by the user is not already stored in the `SEEN_NONCES` storage and is bigger than the smallest nonce in this storage. The relevant code is shown below:

```
SEEN_NONCES.may_update(ctx.storage, |maybe_nonces| {
    let mut nonces = maybe_nonces.unwrap_or_default();

    match nonces.first() {
        Some(&first) => {
            // If there are nonces, we verify the nonce is not yet
            // included as seen nonce and it is bigger than the
            // oldest nonce.
            ensure!(
                !nonces.contains(&metadata.nonce),
                "nonce is already seen: {}",
                metadata.nonce
            );

            ensure!(
                metadata.nonce > first,
                "nonce is too old: {} < {}",
                metadata.nonce,
                first
            );

            // Remove the oldest nonce if max capacity is reached.
            if nonces.len() == MAX_SEEN_NONCES {
                nonces.pop_first();
            }
        },
        None => {
            // Ensure the first nonce is zero.
```

```
                ensure!(metadata.nonce == 0, "first nonce must be 0");
            },
        }

        nonces.insert(metadata.nonce);

        Ok(nonces)
    })?;
```

A user could thus provide any value of `nonce` as long as it is greater than the smallest nonce and is not already present in `SEEN_NONCES`.

Impact

As a user could provide any value of `nonce`, one of the members of the multi-sig account could create transactions with the 20 largest (because `MAX_SEEN_NONCES` is 20) values for `uint32` such that there are no new nonces left. This would lead to a denial of service for other members of the multi-sig as they would not be able to create new transactions because there would not be any acceptable nonce left.

Recommendations

We recommend making sure that the `nonce` is not increased by a large amount.

Remediation

This issue has been acknowledged by Left Curve Software Limited, and a fix was implemented in PR #587 ↗.

3.2. No signature is required in `RegisterUser`

Target	factory		
Category	Business Logic	**Severity**	High
Likelihood	Low	**Impact**	High

Description

To register a username, a user sends an IBC transfer with a deposit with the factory contract's `MINIMUM_DEPOSIT`. Then, they send a transaction containing a single `ExecuteMsg::RegisterUser` message to the factory contract, containing their desired username as well as the `key`, `key_hash`, and `secret` that determines the address for which the deposit is looked up. No signature is required from `key`, despite it being a public key for which only the user has the corresponding private key.

Impact

Anyone observing the `RegisterUser` message before it is accepted into a block (e.g., nodes that front-run, or non-node users if the registration fails due to multiple registrations for the same username in the same block) can submit a `RegisterUser` message with the same `key`, `key_hash`, and `secret` with a different username to spend the user's deposit on an undesired username.

Recommendations

Require a signature of a message containing the desired username.

Remediation

This issue has been acknowledged by Left Curve Software Limited, and a fix was implemented in `PR #583` ↗.

3.3. First nonce is required to be zero

Target	auth		
Category	Business Logic	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

In `verify_nonce_and_signature`, if there are no previously seen nonces, the nonce for the first transaction is required to be zero.

```
None => {
    // Ensure the first nonce is zero.
    ensure!(metadata.nonce == 0, "first nonce must be 0");
},
```

Impact

If multiple transactions are submitted for a new account and arrive out of order, ones that are processed before the one with a nonce of zero will be incorrectly rejected.

Recommendations

Accept nonces in the range `0 ..= MAX_SEEN_NONCES` when `nonces.first()` is `None`.

Remediation

This issue has been acknowledged by Left Curve Software Limited, and a fix was implemented in `PR #592` ↗.

4. System Design

This provides a description of the high-level components of the system and how they interact, including details like a function's externally controllable inputs and how an attacker could leverage each input to cause harm or which invariants or constraints of the system are critical and must always be upheld.

Not all components in the audit scope may have been modeled. The absence of a component in this section does not necessarily suggest that it is safe.

4.1. Component: factory

Description

The factory allows users to register their username, create accounts, and update keys/parameters. There are four execute messages for these tasks.

Message: `ExecuteMsg::RegisterUser`

This message calls the function `register_user` with the provided `username`, `secret`, `key`, and `key_hash`. The function first verifies that the provided username does not already exist and reverts if it already does. Next, it saves the provided key and calls `onboard_new_user`, which generates the address for the account, saves the account address and username, and creates a message to instantiate the account.

Message: `ExecuteMsg::RegisterAccount`

This message calls the function `register_account` with the provided `params`. The function allows users to register an account. It first performs some basic validation to make sure that one can only register accounts for themselves, and it then stores the new derived account address and saves the address. Finally, it creates a message to instantiate the account.

Message: `ExecuteMsg::UpdateKey`

This message calls the function `update_key` with the provided `key_hash` and `key` values. The function is responsible for updating the key for an account. It first finds the username associated with the caller and updates the `KEYS` map.

Message: `ExecuteMsg::UpdateAccount`

This message calls the function `update_account` with the provided `updates`. The function is responsible for updating the account parameters for the multi-sig account.

Invariants

- During user registration, the username must not already exist in the `KEYS` map.
- When a user registers an account, they can only register an account for themselves.
- For multi-sig accounts, the voting threshold should not be greater than total voting power during registration and parameter updates.
- Key updates should only be allowed for a single signature account and not multi-sigs.
- During a key update, there should at least be one key remaining for that username after the update.

Test coverage

Cases covered

- Attempting to register an existing username fails.
- User onboarding and registration works as expected.
- User registration must fail if minimum deposit requirements are not met.
- Key updates must work as expected.

Cases not covered

- N/A.

Attack surface

Users should not be able to register accounts or update keys for others' accounts. This is enforced by making sure that the username belongs to the `ctx.sender` and by making sure that keys are only updated for the username belonging to `ctx.sender`, respectively.

Users should not be able to register a user using a `username` that already exists. This is enforced by verifying that the `KEYS` map does not contain the provided `username` already.

For a multi-sig account, it is important to make sure that the account updates are for the correct account. This is enforced by getting the accounts from the `ACCOUNTS` map for the caller.

4.2. Component: multi

Description

This is the multi-sig account implementation. The following entry points are available for this account type:

1. `instantiate` — can only be called by the account factory and is responsible for instantiating the multi-sig account.
2. `execute` — can only be called by the contract itself.

3. `query` — used to read the state of the contract.

The `authenticate` function is used to perform the authentication for the transactions. The `receive` function accepts incoming transfers.

There are three types of execute messages allowed:

1. `ExecuteMsg::Propose` — used to create a proposal.
2. `ExecuteMsg::Vote` — responsible for voting for/against a proposal.
3. `ExecuteMsg::Execute` — used to execute the proposal if it is passed.

Invariants

- The only type of transaction allowed is to execute itself. Other messages can only be performed through proposals.
- If the action is to vote for a proposal, the voter username in `ExecuteMsg::Vote` must match the signer username in `Metadata`.
- If the action is to vote, the voter/signer must be a member at the time the proposal was created.

Test coverage

Cases covered

- Creating a proposal, voting, and execution are working as expected.
- Manual execution of the proposal is working as expected.
- If the proposal fails due to votes, the status of the proposal should be updated to `Status::Failed`, and it should not be executed.
- Unauthorized voting fails.
- Unauthorized messages should fail.

Cases not covered

- N/A.

Attack surface

The unauthorized voting and message execution should always fail. These cases are well-tested. A user who has already voted should be able to vote again. This case is verified using the following code:

```
ensure!(
    !VOTES.has(ctx.storage, (proposal_id, &voter)),
    "user `{voter}` has already voted in this proposal"
);
```

A proposal that has already been executed should not be allowed to be executed again. This is verified by making sure that the state of the proposal during execution is `Status::Passed` and after the execution it is updated to `Status::Executed`.

4.3. Component: spot

Description

Spot accounts, unlike multi-sig accounts, are controlled by a single user. Their entry points are as follows:

1. `instantiate` — checks that the sender is the factory contract and deducts the deposit.
2. `authenticate` — calls the auth module's `authenticate_tx` (see section 4.4. ↗) to check the user's signature on transactions to execute.
3. `receive` — succeeds unconditionally, approving incoming transfers.
4. `reply` — enforces that the account's current balance exceeds the minimum balance after processing messages.

Invariants

- The account's balance must exceed the specified minimum.

Test coverage

Cases covered

- The factory tests implicitly test the functionality of spot accounts, since spot accounts are the default account type.

Cases not covered

- N/A.

Attack surface

The auth module ensures that transactions are signed directly or indirectly by the owner of the account.

4.4. Component: auth

Description

The auth component consists of functions that the account contracts call to verify that transactions are authorized by the account owner. Transactions contain nonces as a form of replay prevention, but nonces are not required to be strictly sequential to allow multiple in-flight transactions. Note that the nonces used for replay prevention are not the same as the nonces used in the cryptographic signatures: the Secp256k1 signing implementation internally uses RFC 6979 deterministic nonces.

Function: `query_seen_nonces`

The `query_seen_nonces` function returns the current set of `SEEN_NONCES` for the contract, and it is called in response to `QueryMsg::SeenNonces` queries in the margin, multi, and spot account contracts. These queries allow a client to query an account that it manages to determine which nonces to use for future transactions.

Function: `authenticate_tx`

The `authenticate_tx` function is a wrapper around the `verify_nonce_and_signature` function that first queries the factory contract to look up the username associated with the sender of the transaction and then calls `verify_nonce_and_signature`.

Function: `verify_nonce_and_signature`

The `verify_nonce_and_signature` function checks that the transaction's nonce and signature are valid in `Check` and `Finalize` mode and performs no checks in `Simulate` mode.

Function: `verify_signature`

The `verify_signature` function verifies that the provided signature is valid for the specified public key and message. For Secp256k1 keys, the signature can be in either the EIP-712 format or a plain Secp256k1 signature. For Secp256r1 keys, the signature must be in passkey format.

Invariants

- At most one transaction is accepted for each nonce.

Test coverage

Cases covered

- Passkey signatures, EIP-712 signatures, and Secp256k1 signatures are tested with standard credentials.
- Secp256k1 signatures are tested with session credentials.

Cases not covered

- Session credentials are not tested with passkey or EIP-712 signatures.
- The error case of a signature type that does not match a key is not tested.

Attack surface

Function: `verify_nonce_and_signature`

The `verify_nonce_and_signature` function checks that the nonce is not present in `SEEN_NONCES` — it is larger than the smallest nonce in `SEEN_NONCES`. If `SEEN_NONCES` is empty, the transaction's nonce must be zero. It does not currently check that the nonce is not much higher than the largest nonce in `SEEN_NONCES` (see Finding 3.1. ↗). It adds the new nonce to `SEEN_NONCES`, removing the smallest nonce if it would exceed `MAX_SEEN_NONCES` elements.

It checks that if the transaction has an expiry set, the expiry does not exceed the block's timestamp.

The transaction specifies a credential that is either a `Standard` or `Session` credential, both of which have a `key_hash`, which is used to look up the public `key` associated with that `key_hash` and `username`.

For `Session` credentials, it checks that the session credentials' expiry does not exceed the block's timestamp, that the long-term `key` signed the credential's `session_info` (which contains the short-term session key and expiry timestamp), and that the short-term key signed the message authenticating the transaction. For `Standard` credentials, it checks that the user's long-term key signed the message authenticating the transaction.

The message authenticating the transaction, which is either signed directly by the user's long-term key or by the short-term session key, consists of the chain ID and the transaction's nonce, gas limit, sender, messages, and metadata.

5. Assessment Results

At the time of our assessment, the reviewed code was not deployed to the dango mainnet.

During our assessment on the scoped Dango Account and Auth crates, we discovered three findings. No critical issues were found. Two findings were of high impact and the remaining finding was informational in nature.

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.