Zellic

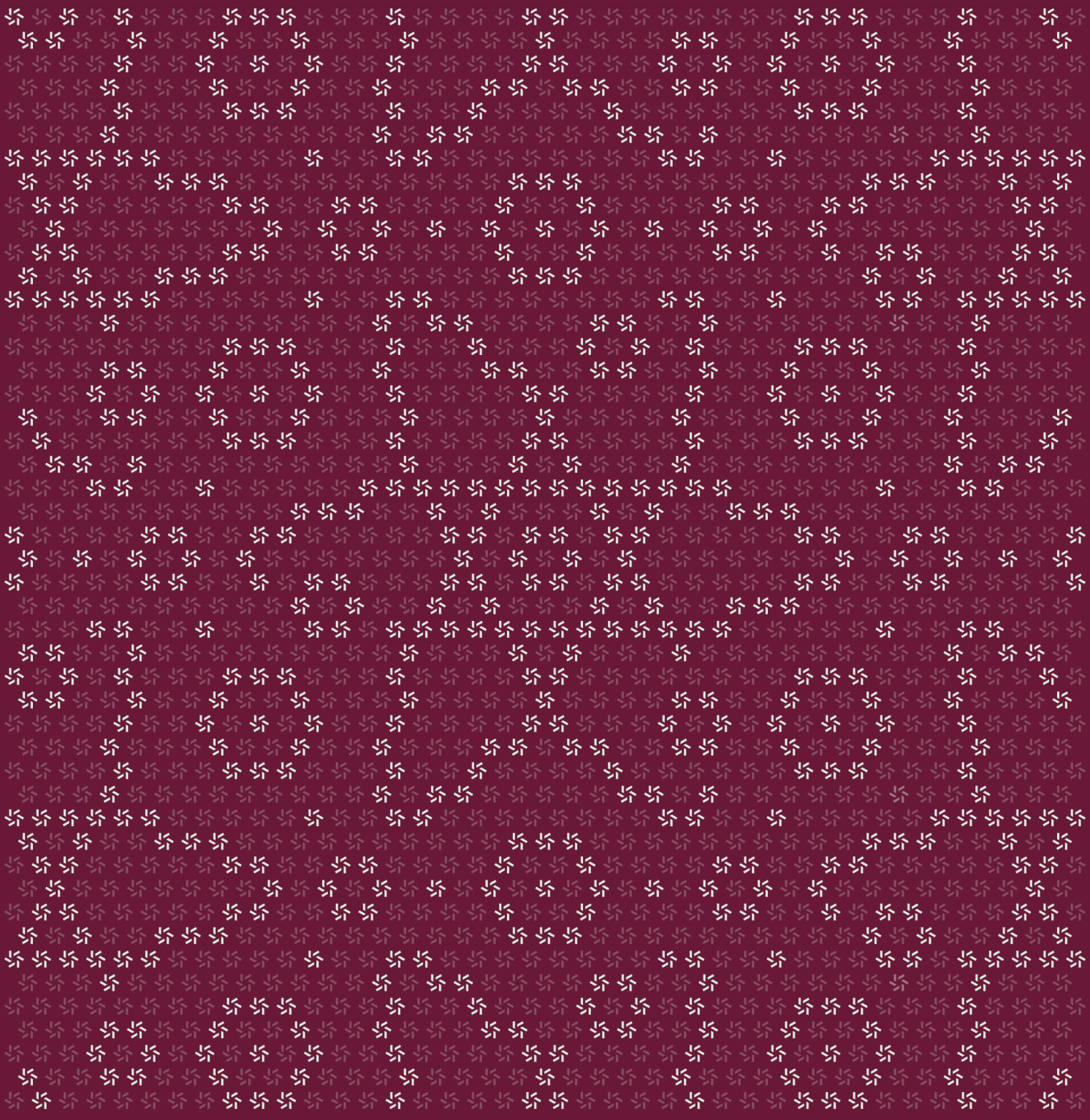
Prepared for
Ash
Fondue
Burnt Labs

Prepared by
Frank Bachman
Jaeeu Kim
Zellic

January 31, 2025

Xion Passkeys

Cosmwasm Contract Security Assessment



Contents

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 Burnt Labs from January 20th to January 23rd, 2025. During this engagement, Zellic reviewed Xion Passkeys'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:

- Could a malicious user DOS or spam the protocol?
- Could a malicious user steal other users' wallets?
- Could a malicious user bypass the authentication mechanism?

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

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

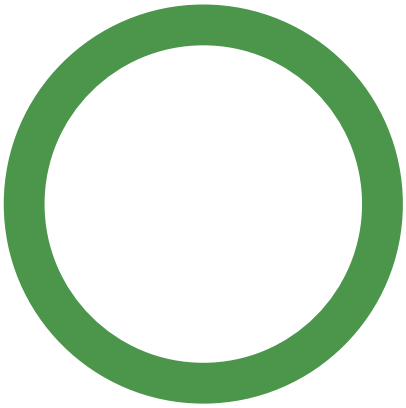
1.4. Results

During our assessment on the scoped Xion Passkeys contracts, we discovered two findings, both of which were low impact.

Additionally, Zellic recorded its notes and observations from the assessment for the benefit of Burnt Labs in the Discussion section (4. ↗).

Breakdown of Finding Impacts

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



2. Introduction

2.1. About Xion Passkeys

Burnt Labs contributed the following description of Xion Passkeys:

> The first walletless blockchain built for mainstream adoption. XION empowers developers to build, launch and scale consumer-ready products from the ground up, removing technical barriers for all users through its Chain Abstraction.

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.

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.

Finally, Zellic provides a list of miscellaneous observations that do not have security impact or are not directly related to the scoped contracts itself. These observations — found in the Discussion (4. ↗) section of the document — may include suggestions for improving the codebase, or general recommendations, but do not necessarily convey that we suggest a code change.

2.3. Scope

The engagement involved a review of the following targets:

Xion Passkeys Contracts

Type	Rust
Platform	cosmwasm-compatible
Target	XION Mainnet Contracts
Repository	https://github.com/burnt-labs/contracts ↗
Version	`5b1dbda8bfc80d9828e94004f19f5822be96d58c`

Programs	`execute.rs`
	`auth/passkey.rs`
	`auth/util.rs`
	`error.rs`
	`lib.rs`
	`query.rs`
	`auth.rs`
	`state.rs`
	`contract.rs`
	`msg.rs`

Target	XION Daemon
Repository	https://github.com/burnt-labs/xion ↗
Version	`5aa867159939b0203421fc0359f2a8d1a7907286`
Programs	`keeper/grpc_query.go` `types/webauthn.go`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of 1.2 person-weeks. The assessment was conducted by two consultants over the course of three 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:

Frank Bachman
Engineer
frank@zellic.io

Jaeeu Kim
Engineer
jaeeu@zellic.io

2.5. Project Timeline

The key dates of the engagement are detailed below.

January 20, 2025	Start of primary review period
January 22, 2025	Kick-off call
January 23, 2025	End of primary review period

3. Detailed Findings

3.1. Bypass overriding index

Target	contracts/account/src/execute.rs		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Medium	**Impact**	Low

Description

The function `add_auth_method` is used to add a new authentication method to the account, and the function `save_authenticator` is used to save `AUTHENTICATORS` with checking the index is not already used to avoid overriding the existing authenticator. However, the `AddAuthenticator::Secp256R1` case in `add_auth_method` does not call `save_authenticator` to save the authenticator, and it directly saves the authenticator to storage.

```
pub fn add_auth_method(
    deps: DepsMut,
    env: &Env,
    add_authenticator: &mut AddAuthenticator,
) -> ContractResult<Response> {
    match add_authenticator.borrow_mut() {
        // ...
        AddAuthenticator::Jwt {
            id,
            aud,
            sub,
            token,
        } => {
            // ...
            save_authenticator(deps, *id, &auth)?;
            Ok(())
        }
        AddAuthenticator::Secp256R1 {
            id,
            pubkey,
            signature,
        } => {
            let auth = Authenticator::Secp256R1 {
                pubkey: (*pubkey).clone(),
            };

            if !auth.verify(
                deps.as_ref(),
```

```
                env,
                &Binary::from(env.contract.address.as_bytes()),
                signature,
            )? {
                Err(ContractError::InvalidSignature)
            } else {
                AUTHENTICATORS.save(deps.storage, *id, &auth)?;
                Ok(())
            }
        }
        AddAuthenticator::Passkey {
            id,
            url,
            credential,
        } => {
            // ...
            save_authenticator(deps, *id, &auth)?;
            // ...
            Ok(())
        }
    }?;
    //...
}

pub fn save_authenticator(
    deps: DepsMut,
    id: u8,
    authenticator: &Authenticator,
) -> ContractResult<()> {
    if AUTHENTICATORS.has(deps.storage, id) {
        return Err(ContractError::OverridingIndex { index: id });
    }

    AUTHENTICATORS.save(deps.storage, id, authenticator)?;
    Ok(())
}
```

Impact

This could lead to overriding the existing authenticator in storage. There is no security impact on the contract, but if both add and remove auth method events occur in a tracking scenario, it could lead to confusion.

Recommendations

Change the `AddAuthenticator::Secp256R1` case in `add_auth_method` to call `save_authenticator` to save the authenticator.

Remediation

This issue has been acknowledged by Burnt Labs, and a fix was implemented in commit `0e1583eb` ↗.

3.2. Malformed Ether address could be added

Target	contracts/account/src/auth.rs		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Medium	**Impact**	Low

Description

The function `verify` is used to verify the signature of the transaction. However, the code decodes the hex string from index 2 of the address (skipping the "0x" prefix), effectively ignoring the first two characters. The two bytes of `address` in `Authenticator::EthWallet` are not checked for the correct format of the Ethereum address. The `address` is expected to be a valid Ethereum address, and the address should be 42 characters long and start with `0x`.

In this case, the first two bytes could be any value if last 40 bytes are a valid Ethereum address. This could lead to a malformed Ethereum address being added as an authenticator.

```
pub fn add_auth_method(
    deps: DepsMut,
    env: &Env,
    add_authenticator: &mut AddAuthenticator,
) -> ContractResult<Response> {
    // ...
        AddAuthenticator::EthWallet {
            id,
            address,
            signature,
        } => {
            let auth = Authenticator::EthWallet {
                address: (*address).clone(),
            };

            if !auth.verify(
                deps.as_ref(),
                env,
                &Binary::from(env.contract.address.as_bytes()),
                signature,
            )? {
                Err(ContractError::InvalidSignature)
            } else {
                save_authenticator(deps, *id, &auth)?;
                Ok(())
```

```
            }
        }

// contracts/account/src/auth.rs
impl Authenticator {
    pub fn verify(
        &self,
        deps: Deps,
        env: &Env,
        tx_bytes: &Binary,
        sig_bytes: &Binary,
    ) -> Result<bool, ContractError> {
        // ...
            Authenticator::EthWallet { address } => {
                let addr_bytes: Vec<u8> = hex::decode(&address[2..])?;
                match eth_crypto::verify(deps.api, tx_bytes, sig_bytes,
    &addr_bytes) {
                    Ok(_) => Ok(true),
                    Err(error) => Err(error),
                }
            }
```

Impact

A malformed Ethereum address could be added as an authenticator, such as `AAc02aaa39b223fe8d0a0e5c4f27ead9083c756cc2` or `??c02aaa39b223fe8d0a0e5c4f27ead9083c756cc2`.

This could confuse the tracker who is tracking the authenticator's address.

Recommendations

Check the address for the correct format of the Ethereum address before adding it as an authenticator.

Remediation

This issue has been acknowledged by Burnt Labs, and a fix was implemented in commit `0e1583eb` ↗.

4. Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment. These discussion notes are not necessarily security related and do not convey that we are suggesting a code change.

4.1. Test suite

Here we note two areas for test-suite improvement.

Lack of unit tests. For the cosmwasm contract, a test suite could be easily implemented. A test suite helps to catch the unexpected behavior of a contract. Currently, many parts of the contract are not covered by a test suite. Good test coverage has multiple effects.

- It finds bugs and design flaws early (preaudit or prerelease).
- It displays code maturity.
- It bolsters customer trust in your product.
- It improves understanding of how the code functions, integrates, and operates — for developers and auditors alike.
- It increases development velocity long-term.

5. Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the contracts and created a written threat model for some critical functions. 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.

5.1. Endpoints

`instantiate`

The `instantiate` is expected to be called by the owner of the contract. This calls `execute::init`, and it calls `add_auth_method` with the first authenticator, which is the owner of the contract.

`execute`

The `execute` is allowed to be called by anyone. However, `execute` asserts that the caller is the contract itself. So, it is not possible to call this entry point from the outside by someone who is not the owner of the contract.

`AddAuthMethod`

This adds the authenticator to storage. A new authenticator can be added when verifying is successful, so an invalid authenticator cannot be added to storage.

Controllable parameters

- `add_authenticator`: The authenticator to add.

The authenticator type could be one of the following:

- Secp256K1
  - `id`: The ID of the authenticator.
  - `pubkey`: The public key of the authenticator.
  - `signature`: The signature of the authenticator.
- Ed25519
  - `id`: The ID of the authenticator.
  - `pubkey`: The public key of the authenticator.
  - `signature`: The signature of the authenticator.
- EthWallet
  - `id`: The ID of the authenticator.
  - `address`: The address of the authenticator.

    - `signature`: The signature of the authenticator.
- Jwt
    - `id`: The ID of the authenticator.
    - `aud`: The audience of the authenticator.
    - `sub`: The subject of the authenticator.
    - `token`: The token of the authenticator.
- Secp256R1
    - `id`: The ID of the authenticator.
    - `pubkey`: The public key of the authenticator.
    - `signature`: The signature of the authenticator.
- Passkey
    - `id`: The ID of the authenticator.
    - `url`: The URL of the authenticator.
    - `credential`: The credential of the authenticator.

`RemoveAuthMethod`

This removes the authenticator from storage using an ID. It only can be called when the authenticator count is bigger than 2. Therefore, since there is at least one authenticator present, it cannot be a contract with restricted access.

Controllable parameters

- `id`: The ID of the authenticator to remove.

`Emit`

This writes an event with a message that is not bigger than 1,024 bytes.

Controllable parameters

- `data`: The message to write in the event.

`sudo`

The `sudo` is called by the chain itself, so it will be triggered by XION chain.

`AccountSudoMsg::BeforeTx`

The `AccountSudoMsg::BeforeTx` is called before a transaction is executed. It is used to check if the transaction is valid or not using verification. It hashes the original transaction bytes and compares

it with the signature using the authenticator.

Controllable parameters

- `tx_bytes`: The original transaction bytes.
- `cred_bytes`: The credential bytes. The first byte will be the index of the authenticator, and other bytes will be the signature, which is used in verifying.
- `simulate`: The flag to check if the transaction is simulated or not.

6. Assessment Results

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

During our assessment on the scoped Xion Passkeys contracts, we discovered two findings, both of which were low impact.

6.1. 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.