Audit Report

CosmWasm

v1.0

March 27, 2023

Table of Contents

License

Disclaimer

This audit has been performed by

Oak Security

https://oaksecurity.io/
info@oaksecurity.io

Introduction

Purpose of This Report

Oak Security has been engaged by Confio GmbH to perform a security audit of CosmWasm.

The objectives of the audit are as follows:

1. Determine the correct functioning of the protocol, in accordance with the project specification.
2. Determine possible vulnerabilities, which could be exploited by an attacker.
3. Determine smart contract bugs, which might lead to unexpected behavior.
4. Analyze whether best practices have been applied during development.
5. Make recommendations to improve code safety and readability.

This report represents a summary of the findings.

As with any code audit, there is a limit to which vulnerabilities can be found, and unexpected execution paths may still be possible. The author of this report does not guarantee complete coverage (see disclaimer).

Codebase Submitted for the Audit

The audit has been performed on the following GitHub repository:

https://github.com/cosmwasm/cosmwasm

The following directories were in scope:

- packages/crypto
- packages/derive
- packages/schema
- packages/std
- packages/storage
- packages/vm

Commit hash: `a0cf296c43aa092b81457d96a9c6bc2ab223f6d3`

Methodology

The audit has been performed in the following steps:

1. Gaining an understanding of the code base's intended purpose by reading the available documentation.
2. Automated source code and dependency analysis.
3. Manual line by line analysis of the source code for security vulnerabilities and use of best practice guidelines, including but not limited to:
    a. Race condition analysis
    b. Under-/overflow issues
    c. Key management vulnerabilities
4. Report preparation

Functionality Overview

The audited code implements the CosmWasm smart contract standard library (consisting of bindings and imports, convenience helpers for storage interaction), the entrypoint macro, a wrapper around Wasmer (which includes gas metering, callbacks, address transformations), cryptographic functions, and a schema generator.

How to Read This Report

This report classifies the issues found into the following severity categories:

Severity	Description
Critical	A serious and exploitable vulnerability that can lead to loss of funds, unrecoverable locked funds, or catastrophic denial of service.
Major	A vulnerability or bug that can affect the correct functioning of the system, lead to incorrect states or denial of service.
Minor	A violation of common best practices or incorrect usage of primitives, which may not currently have a major impact on security, but may do so in the future or introduce inefficiencies.
Informational	Comments and recommendations of design decisions or potential optimizations, that are not relevant to security. Their application may improve aspects, such as user experience or readability, but is not strictly necessary. This category may also include opinionated recommendations that the project team might not share.

The status of an issue can be one of the following: **Pending,** **Acknowledged** or **Resolved**.

Note that audits are an important step to improve the security of smart contracts and can find many issues. However, auditing complex codebases has its limits and a remaining risk is present (see disclaimer).

Users of the system should exercise caution. In order to help with the evaluation of the remaining risk, we provide a measure of the following key indicators: **code complexity**, **code readability**, **level of documentation**, and **test coverage**. We include a table with these criteria below.

Note that high complexity or low test coverage does not necessarily equate to a higher risk, although certain bugs are more easily detected in unit testing than a security audit and vice versa.

Summary of Findings

No	Description	Severity	Status
1	Cache hit counters can be used to crash nodes and halt a chain	Critical	Resolved
2	Gas counting in vm backend does not correctly add externally used gas	Minor	Resolved
3	Overflow in gas counting might halt block production	Minor	Acknowledged
4	Ed25519 batch verification is not benchmarked, might run out of memory and halt block production	Minor	Acknowledged
5	Unlimited WASM table size may be exploited to crash the node	Minor	Resolved
6	Unbounded iteration over WASM imports may slow down block production or stop the chain	Minor	Resolved
7	Compilation of wasm code does not specify memory limit and might halt block production	Informational	Acknowledged
8	Attribute keys starting with underscores lead to panics, causing smart contract runtime errors in debug mode	Informational	Acknowledged
9	Overflows could occur if library users do not enable overflow-checks, panics abort execution	Informational	Acknowledged
10	Math method types are inconsistent and not exhaustive	Informational	Acknowledged
11	Several dependencies are outdated	Informational	Resolved
12	Unaudited cryptography library	Informational	Acknowledged
13	Ed25519 batch verification succeeds for empty data, which may not be obvious to library users	Informational	Acknowledged
14	Consuming gas after performing Ed25519 batch verification is inefficient	Informational	Resolved
15	Missing overflow checks	Informational	Acknowledged

Code Quality Criteria

Criteria	Status	Comment
Code complexity	Medium-High	Among other features, the packages audited implement bindings between a WebAssembly host and guest, including memory management. Naturally, this leads to high complexity.
Code readability and clarity	Medium-High	-
Level of Documentation	Medium-High	User-facing documentation is present and exhaustive. One area that could be improved is the gas counting and cost, for example for repeated storage access. The code is commented well, albeit higher level architectural documentation could be improved, especially on the interaction of the various components.
Test Coverage	Medium-High	-

Detailed Findings

1. Cache hit counters can be used to crash nodes and halt a chain

Severity: Critical

The VM implementation contains a cache to efficiently load smart contract instances that are pinned or used repeatedly. For analytical purposes, that cache increments the hit counters `hits_pinned_memory_cache`, `hits_memory_cache`, and `hits_fs_cache` in `packages/vm/src/cache.rs` whenever an instance is read from it. These hit counters are incremented using integer addition, which can cause panics in the case of integer overflows.

An attacker can exploit this issue to crash a node and possibly even halt the chain, for example by repeatedly querying an instance from a CosmWasm smart contract.

The hit counters currently use the `u32` type. To cause an overflow, almost 4.3 billion cash hits are required. While this is a huge number, the counters are only reset upon node restart, so there is a real possibility for long-running nodes to suffer from such an overflow panic. Moreover, an attacker does not need to cause almost 4.3 billion cache hits within one transaction – such an attack can be executed across many blocks. Query nodes are even easier to target, since they can be queried repeatedly with little resource cost by the attacker.

Recommendation

We recommend using saturating addition rather than normal integer addition for cache hit counters.

Status: Resolved

2. Gas counting in vm backend does not correctly add externally used gas

Severity: Minor

The `GasInfo`'s `add_assign` method incorrectly sets in `packages/vm/src/backend.rs:56`:

```
externally_used: self.externally_used + other.cost
```

This implies that the `externally_used` gas of `other` is not accounted for. An attacker could call external functions without paying gas for them, potentially blocking block production.

We classify this issue as minor since the `add_assign` method is currently only used in test code.

Recommendation

We recommend changing `packages/vm/src/backend.rs:56` to:

```
externally_used: self.externally_used + other.externally_used
```

Status: Resolved

3. Overflow in gas counting might halt block production

Severity: Minor

The `add_asign` and `process_gas_info` functions might overflow in `packages/vm/src/backend.rs:55`, `56`, `packages/vm/src/environment.rs:341` and `352`, which could potentially allow an attacker to halt block production of the underlying blockchain if the CosmWasm vm is compiled without `overflow-checks` enabled for the release profile.

Recommendation

We recommend using `checked_add` in `packages/vm/src/backend.rs:55`, `56` as well as in `packages/vm/src/environment.rs:341` and `352`.

Status: Acknowledged

The CosmWasm team states: “CosmWasm gas aims for 1 Teragas/millisecond, i.e. the uint64 range exceeds after 18 million seconds (5 hours). Assuming a max supported block execution time of 30 seconds, the gas price has to be over-priced by a factor of 614 (614 Teragas/millisecond) in order to exceed the uint64 range. Since serious over or underpricing is considered a bug, using uint64 for gas measurements is considered safe.” Additionally, the CosmWasm team plans to adjust the gas price in CosmWasm 2 for an additional 1000x safety margin, see https://github.com/CosmWasm/cosmwasm/issues/1599 for details.

4. Ed25519 batch verification is not benchmarked, might run out of memory and halt block production

Severity: Minor

The `do_ed25519_batch_verify` function imposes limits to the number of messages, signatures, and public keys per batch in `packages/vm/src/imports.rs:291-305`, but there currently exists no benchmarking code in the codebase that verifies that these limits are

reasonable. If the limits are too high, validators might run out of memory, and block production of the underlying blockchain might halt.

Recommendation

We recommend running benchmarks to determine sensible values for Ed25519 batch verification limits.

Status: Acknowledged

5. Unlimited WASM table size may be exploited to crash the node

Severity: Minor

While the memory of instances is limited in `packages/vm/src/wasm_backend/store.rs:82`, tables are not. The WebAssembly spec states that if tables are not limited, they can grow indefinitely (see https://webassembly.github.io/spec/core/syntax/types.html#table-types). If such a growth of a table can be triggered from smart contracts, it may exhaust the resources of the underlying node, potentially crashing it and stopping block production.

We tried to write a WASM module using WAT that adds new elements in a loop, but apparently, `elem` sections need to be at the module top level, so adding more entries to the table would linearly increase the WASM blob size.

While we have not been able to exploit this issue by adding elements to the table, there may be other ways to make the table size grow indefinitely.

Recommendation

We recommend limiting the table size.

Status: Resolved

6. Unbounded iteration over WASM imports may slow down block production or stop the chain

Severity: Minor

The `check_wasm_imports` function collects all imports of the wasm contract into a `BTreeSet` in `packages/vm/src/compatibility.rs:152`, which uses an unbounded iteration over the imports. This may be exploited to slow down block production, possibly even stopping the chain.

We classify this issue as minor since there is an implicit limit to the number of imports through the WASM size limit.

Recommendation

We recommend limiting the number of iterations when checking wasm imports.

Status: Resolved

7. Compilation of wasm code does not specify memory limit and might halt block production

Severity: Informational

The `save_wasm` function calls `compile` in `packages/vm/src/cache.rs:154` without setting a memory limit. This could cause the node to run out of memory, potentially halting block production of the underlying blockchain if no mechanism for recovery is implemented.

We only consider this to be an informational issue since the underlying blockchain should reject wasm code that is too big, and the singlepass compiler used will execute in linear time, preventing JIT-bombs.

Recommendation

We still recommend setting a sensible memory limit as a precaution.

Status: Acknowledged

8. Attribute keys starting with underscores lead to panics, causing smart contract runtime errors in debug mode

Severity: Informational

The constructor for attributes panics in debug mode in `packages/std/src/results/events.rs:71` if an attribute key starts with an underscore _. This will lead to runtime errors for smart contract projects that have code paths that were not extensively tested. An example could be a DeFi protocol that adds an attribute with an underscore under certain conditions, for example an emergency withdrawal. The panic would prevent the emergency withdrawal, putting user funds at risk.

The check in the constructor does also not guarantee that a key does not start with an underscore, since library users could simply manually push attributes with any key.

We classify this issue as informational since it only affects code compiled with debug assertions.

Recommendation

We recommend escaping/adding a prefix to keys starting with an underscore and returning an error if debug mode is enabled that explains the prefix, rather than panicking. We also recommend clearly documenting this behavior.

Status: Acknowledged

9. Overflows could occur if library users do not enable overflow-checks, panics abort execution

Severity: Informational

In several places in the codebase, overflows are not handled explicitly. This is not a security concern as long as overflow-checks are enabled for the release profile, but there is a risk that a library user does not enable overflow-checks. Additionally, overflow-checks cause overflows to panic, which is less user friendly than an overflow error message that allows unwinding the operation. A panic in WebAssembly always aborts the execution and does not unwind. Instances of potential overflows are:

- The `nextval` function in `packages/storage/src/sequence.rs:19`.
- The `plus_seconds`, `plus_nanos`, `minus_seconds`, and `minus_nanos` methods in `packages/std/src/timestamp.rs:37-57`.
- The `isqrt` method in `packages/std/src/math/isqrt.rs:28`.
- The `sqrt_with_precision` method in `packages/std/src/math/decimal.rs:181` and in `packages/std/src/math/decimal256.rs:194`.

Recommendation

We recommend using the `checked_*` functions and returning overflow error messages if overflows occur.

Status: Acknowledged

The CosmWasm team states that every contract must have overflow-checks enabled in their release profile, and intends to clearly document this in guidelines as well as implement it as a CosmWasm linter rule.

10. Math method types are inconsistent and not exhaustive

Severity: Informational

In `packages/std/src/math`, several math types are implemented, such as `Decimal`, `Decimal256`, `Uint64`, `Uint128`, or `Uint256`. The API of those math types contains several inconsistencies and functionality diverges or is partially not implemented, even between similar types such as `Uint64` and `Uint128`. These differences can lead to a decrease in the development experience when using math types of the std package.

Examples of these discrepancies are:

- `Sub` and `Mul` are implemented for `Uint128`, `Uint256` and `Uint512`, but missing for `Uint64`.
- `SubAssign` and `MulAssign` are implemented for `Uint128`, `Uint256` and `Uint512`, but missing for `Uint64`.
- `wrapping_add`, `wrapping_sub`, `wrapping_mul`, and `wrapping_pow` are implemented for `Uint64` and `Uint128`, but missing for `Uint256` and `Uint512`.
- `Mul` is implemented for `Decimal` and `Decimal256`, but `Div` is not implemented for them.
- For `Decimal` and `Decimal256`, `Div` returns a `Decimal`, while `DivAssign` returns a `Uint128`
- `multily_ratio` and `full_mul` is implemented for `Uint64`, `Uint128` and `Uint256` but missing for `Uint512`.
- `pow` is implemented for `Uint256` but missing for `Uint64`, `Uint128` and `Uint512`.
- `checked_pow` is implemented for `Uint128` and `Uint256` but missing for `Uint64` and `Uint512`.
- `saturating_pow` is implemented for `Uint128`, but missing for `Uint256` and `Uint512`.
- `check_div_euclid` is implemented for `Uint64` and `Uint128`, but missing for `Uint256` and `Uint512`.
- `Shl` is implemented for `Uint256`, but missing for `Uint64`, `Uint128` and `Uint512`.
- `ShrAssign` is implemented for `Uint64`, `Uint128`, `Uint256` and `Uint512`, but `ShlAssign` is missing for all unsigned integers.
- `checked_shl` is implemented for `Uint256`, but missing for `Uint64`, `Uint128` and `Uint512`.
- Inconsistent with other methods, `TryFrom<Uint128> for Uint64` is defined in `packages/std/src/math/uint128.rs`, and not in `packages/std/src/math/uint64.rs`.
- Inconsistent with other methods, `TryFrom<Uint512> for Uint128` is defined in `packages/std/src/math/uint512.rs`, and not in `packages/std/src/math/uint128.rs`.
- The `from_str` function results in an error on an empty string in `packages/std/src/math/uint256.rs`, but not in `packages/std/src/math/uint128.rs` and `packages/std/src/math/uint512.rs`, which is inconsistent.

- Inconsistent with other methods, `TryFrom<Uint256> for Uint128` is defined in `packages/std/src/math/uint256.rs`, and not in `packages/std/src/math/uint128.rs`.
- In `packages/std/src/math/uint128.rs:215`, `StdError::generic_err` is used for parsing errors instead of `StdError::parse_err`.
- The `new` function in `packages/std/src/math/uint512.rs` is not marked as const, which it is in `packages/std/src/math/uint256.rs`.
- Some methods return `StdError::generic_err("Error parsing whole")` while others return an `ParseErr`.
- Some methods return `StdError::generic_err(format!("Serializing QueryRequest: {}", serialize_err))` instead of `StdError::serialize_err`.
- The `wasmer::SerializeError` and `wasmer::DeserializeError` is wrongly transformed into a `VmError::cache_err` instead of a `VmError::serialize_err` in `packages/vm/src/errors/vm_error.rs:325` and `331`.
- The `VmError::cache_err` in `packages/vm/src/checksum.rs:47` should be a `VmError::conversion_err`.
- Some methods use `expect`, while others use `unwrap`, which results in panics without messages explaining what went wrong.

Recommendation

We recommend providing consistent and exhaustive functions across math types, for example by defining traits that are implemented across types.

Status: Acknowledged

The CosmWasm team states that math methods are currently reworked and progress is tracked in https://github.com/CosmWasm/cosmwasm/issues/1186.

11. Several dependencies are outdated

Severity: Informational

There are several dependencies that are not used in their latest version. Some of them contain security vulnerabilities. A subset that might introduce vulnerabilities is:

- `chrono` contains a potential segfault, currently used in version `0.4.19`, no safe upgrade available.
- `time` contains a potential segfault, currently used in version `0.1.43`, upgrade to >= `0.2.23` instead.
- `ed25519-zebra` is currently used in version `2.2.0`, available in `3.0.0`
- `rand_core` is currently used in version `0.5.1` in package `cosmwasm-crypto`, available in `0.6.3`

Recommendation

We recommend running `cargo audit` and `cargo outdated` to identify outdated and vulnerable dependencies and updating those to the latest stable version.

Status: Resolved

12. Unaudited cryptography library

Severity: Informational

The library implementation of ECDSA in crate `k256` is unaudited and comes with the following warning in its documentation:

Security Warning

The elliptic curve arithmetic contained in this crate has never been independently audited!

This crate has been designed with the goal of ensuring that secret-dependent operations are performed in constant time (using the subtle crate and constant-time formulas). However, it has not been thoroughly assessed to ensure that generated assembly is constant time on common CPU architectures.

USE AT YOUR OWN RISK!

Recommendation

We recommend using audited cryptography libraries whenever possible. If this is not possible for a particular cryptography primitive, the latest version should be used and an end-user warning should be included.

Status: Acknowledged

The CosmWasm team acknowledges this issue, stating that across the industry, typically cryptography libraries are written by academic researchers, but most of them are not audited.

13. Ed25519 batch verification succeeds for empty data, which may not be obvious to library users

Severity: Informational

The `ed25519_batch_verify` function in `packages/crypto/src/ed25519.rs:65` succeeds if provided with an empty collection of messages, signatures, and public keys. This

behavior is correctly documented in line `64`. Still, an unsuspecting library user might not be aware of this implementation, which could lead to bugs in dependent codebases.

Recommendation

While there is no definitive correct behavior, we recommend protecting library users as much as possible by failing for empty slices.

Status: Acknowledged

14. Consuming gas after performing Ed25519 batch verification is inefficient

Severity: Informational

The `do_ed25519_batch_verify` function currently consumes gas in `packages/vm/src/imports.rs:312-318` after executing the batch verification in line `311`. This is inefficient since the verification can be skipped if the gas limit is hit.

Recommendation

We recommend moving the gas consumption above executing the batch verification.

Status: Resolved

15. Missing overflow checks

Severity: Informational

The workspace manifest `Cargo.toml` as well as the following packages do not enable `overflow-checks` for the release profile:

- `packages/crypto/Cargo.toml`
- `packages/derive/Cargo.toml`
- `packages/profiler/Cargo.toml`
- `packages/schema/Cargo.toml`
- `packages/std/Cargo.toml`
- `packages/storage/Cargo.toml`
- `packages/vm/Cargo.toml`

Recommendation

We recommend enabling overflow checks in all packages, including those that do not currently perform calculations, to prevent unintended consequences if changes are added in future releases or during refactoring. Note that enabling overflow checks in packages other than the workspace manifest will lead to compiler warnings.

Status: Acknowledged