Audit Report

wasmvm

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 the wasmvm repository.

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/wasmvm

Commit hash: `3f457fc2f7371bf32f8ae057c53bf39ead08c9ac`

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 wasmvm codebase provides a wrapper around the CosmWasm's `packages/vm` which is written in Rust to allow compilation, initialization and execution from Go applications. That enables integration of the vm in applications such as the Cosmos SDK's `x/wasm` module.

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	Unlimited iterator stack might allow an attacker to crash the node, halting block production	Critical	Resolved
2	FFI result handling may lead to memory leaks in certain cases	Major	Resolved
3	Gas overflow errors are treated as normal panics and do not consume Cosmos SDK gas, which can be exploited to halt block production	Minor	Resolved
4	IBC packet receive function's result is not unwrapped, which is inconsistent and error-prone	Minor	Acknowledged
5	Assumption that caller of VM's create function limits contract size is inconsistent and error-prone	Informational	Acknowledged
6	Caught Rust panics do not log errors	Informational	Resolved
7	Outdated specification/documentation	Informational	Acknowledged
8	Inconsistent type usage for `usedGas`	Informational	Resolved

Code Quality Criteria

Criteria	Status	Comment
Code complexity	Medium-High	-
Code readability and clarity	Medium-High	-
Level of documentation	Medium	-
Test coverage	Medium-High	-

Detailed Findings

1. Unlimited iterator stack might allow an attacker to crash the node, halting block production

Severity: Critical

The global map `iteratorStack` in `api/iterator.go` has no upper limit, which might allow an attacker to exhaust the memory of the node by creating recursive CosmWasm messages/queries that create many entries on the stack. While gas limits might prevent this issue if they are set properly, there is no guarantee that the memory is big enough to hold all the iterator stack entries the gas limit permits. This issue might cause the node to crash, potentially leading to a halt of block production if block producers are affected.

Recommendation

We recommend adding a maximum query/message call stack depth as well as benchmarking the gas consumption of recursive messages/queries to determine adequate limits.

Status: Resolved

2. FFI result handling may lead to memory leaks in certain cases

Severity: Major

During calls from Rust to Go code through the FFI, `UnmanagedVector`s are created for error messages. This happens in:

- The `canonical_address` function in `libwasmvm/src/api.rs:51`.
- The `human_address` function in `libwasmvm/src/api.rs:79`.
- The `next` function in `libwasmvm/src/iterator.rs:60`.
- The `query_raw` function in `libwasmvm/src/querier.rs:45`.
- The `get` function in `libwasmvm/src/storage.rs:29`.
- The `scan` function in `libwasmvm/src/storage.rs:65`.
- The `set` function in `libwasmvm/src/storage.rs:118`.
- The `remove` function in `libwasmvm/src/storage.rs:146`.

These `UnmanagedVector`s are destroyed within the `into_ffi_result` function, but only if the closure defined in `libwasmvm/src/error/go.rs:75-80` is executed. That only happens in the cases where the `GoResult` is of either the `GoResult::Other` or `GoResult::User` variant. In all other cases, a memory leak can occur if the called code is writing to the `UnmanagedVector`s.

In some cases, a similar issue exists with the result coming from through the FFI, for instance:

- The `canonical_address` function in `libwasmvm/src/api.rs:50`.

- The `human_address` function in `libwasmvm/src/api.rs:78`.
- The `next` function in `libwasmvm/src/iterator.rs:58` and `59`.
- The `query_raw` function in `libwasmvm/src/querier.rs:44`.
- The `get` function in `libwasmvm/src/storage.rs:28`.

Those `UnmanagedVector`s are only consumed if no error is returned, but the Go code could in theory write to those unmanaged vectors, even if an error is set. This would also cause a memory leak.

Finally, the iterator's `next` function only consumes the `UnmanagedVector` of the `value` if the `key` is not `None` in `libwasmvm/src/iterator.rs:83`.

Recommendation

We recommend consuming `UnmanagedVector`s in all cases in the `into_ffi_result` function, independent of the returned error value and independent of the iterator's `key` Option.

Status: Resolved

3. Gas overflow errors are treated as normal panics and do not consume Cosmos SDK gas, which can be exploited to halt block production

Severity: Minor

Within callbacks into Cosmos SDK, there is a special `ErrorGasOverflow` which occurs if the uint64 that is used to store gas consumption overflows (see `store/types/gas.go` in Cosmos SDK). In such a case of an `ErrorGasOverflow`, Cosmos SDK sets the consumed gas to 0 in `store/types/gas.go:77`.

The wasmvm callbacks treat such gas overflow errors as normal panics, as opposed to treating them as `ErrorOutOfGas`, see `api/callbacks.go:73-78`. This leads to a gas consumption of 0 units for the Cosmos SDK related functionality.

An attacker may exploit this by sending multiple messages that deliberately cause the Cosmos SDK gas counting to overflow. While such messages would revert, they would consume less gas than expected such that block production might surpass Tendermint's propose timeout. That could cause block production to halt.

Recommendation

We recommend treating the case of a gas overflow in `api/callbacks.go:73` like an `ErrorOutOfGas`, which will lead to gas exhaustion in `x/wasm/keeper/connector.go:107`.

Status: Resolved

The CosmWasm team pointed out that for this issue to become a problem in practice, either the max gas limit needs to be set close to the maximum uin64 value (`2**64-1`) or a massive gas amount close to the maximum uint64 value needs to be consumed in a single `ConsumeGas` call. Both conditions are highly unlikely to be met under normal conditions – the Cosmos Hub for example has a gas limit of 1.5 million, and consumes around 100-300 nanoseconds per gas. It could still happen though, for example due to a bug consuming massive amounts of gas. We therefore classify this issue as minor.

4. IBC packet receive function's result is not unwrapped, which is inconsistent and error-prone

Severity: Minor

The `IBCPacketReceive` function in `lib.go:510` returns a result, and does not unwrap the errors as all other functions in `lib.go` do. That is inconsistent and may lead to errors if not properly handled by the calling context.

Recommendation

We recommend unwrapping the result in as is done in all other functions, i.e. by replacing line `545` with:

```
if resp.Err != "" {
      return nil, gasUsed, fmt.Errorf("%s", resp.Err)
}
return resp.Ok, gasUsed, nil
```

Status: Acknowledged

The client plans to resolve this issue in the future, the issue is tracked at https://github.com/CosmWasm/wasmvm/issues/398.

5. Assumption that caller of VM's create function limits contract size is inconsistent and error-prone

Severity: Informational

As mentioned in the TODO in `lib.go:67`, the VM's `Create` function does currently not enforce any gas limits during contract creation. Since the singlepass compiler is used, gas counting during compilation is not necessary – but the caller of the function should ensure that the size of the wasm code is limited such that block production of the underlying blockchain cannot come to a halt.

Moving that responsibility to the caller is inconsistent since other functions of the VM do accept gas limits. It also requires clear documentation and introduces the likelihood of mistakes by the wasmvm integrator.

Recommendation

We recommend adding a size or gas limit parameter to the `Create` function. While this is functionally equivalent to limiting the size or gas in the calling context, it is more consistent with other VM functions and reduces the risk of integration mistakes.

Status: Acknowledged

The client plans to resolve this issue in the future, the issue is tracked at https://github.com/CosmWasm/wasmvm/issues/395.

6. Caught Rust panics do not log errors

Severity: Informational

In several places in the codebase, panics are caught using `catch_unwind`. `catch_unwind` returns an `Err(cause)`, where `cause` is the object that invoked the panic. Currently, the cause is neither logged nor returned. This negatively impacts maintainability.

Instances are: `libwasmvm/src/cache.rs:43`, `92`, `116`, `143`, `169`, `230`, `305`, as well as `libwasmvm/src/calls.rs:433` and `517`.

Recommendation

We recommend returning or logging the `cause` object.

Status: Resolved

7. Outdated specification/documentation

Severity: Informational

The documentation in various Markdown files is outdated, for example:

- `spec/Specification.md:43` describes that the balance of a contract instance's account is passed in, which is not implemented at the moment. Accordingly, the current implementation of the `ContractInfo` in `types/env.go:24` does not contain the `Balance` field as is documented in lines `99-100`.
- The documentation of the `Instantiate`, `Execute` and `Query` interfaces in `spec/Specification.md:54-59` do not match their implementation in `lib.go:118`, `166`, and `211`.
- The documentation of the `Result` struct in `spec/Specification.md:116` does not match the implementation of `ContractResult` in `types/msg.go:12`.
- The documentation of the `CosmosMsg` struct in `spec/Specification.md:141` does not match its implementation in `types/msg.go:68`.
- The documented `ContractMsg` and `OpaqueMsg` structs in `spec/Specification.md:161` and `180` are not implemented. Also in lines `spec/Specification.md:159` and `types/msg.go:238-239` it is stated that a contract is immutable once deployed, but there is no enforcement of immutable interfaces between contract upgrades.
- The `Params` struct in `spec/Specification.md:75` is now called `Env` and does contain `TransactionInfo` instead of `MessageInfo`, see `types/env.go`.
- The comment in `spec/Specification.md:202` describes the possibility of providing storage iteration/scans in the future, which is already implemented.
- Additionally, the codebase contains several `TODO`s in comments that are already resolved, e. g. in `types/ibc.go:148` or in `lib.go:67` (gas counting is addressed by caller and singlepass backend prevents JIT bombs in the contract code).

Recommendation

We recommend updating the documentation.

Status: Acknowledged

The client states that this issue is partially tracked in https://github.com/CosmWasm/wasmvm/issues/39.

8. Inconsistent type usage for `usedGas`

Severity: Informational

The type of `usedGas` is used inconsistently – in some instances `C.uint64` is used, in others the type alias `cu64`: The functions `cSet`, `cDelete`, and `cQueryExternal` utilize `C.uint64` instead of `cu64` in `api/callbacks.go`.

Recommendation

We recommend consistently using the `cu64` alias for higher readily and code clarity.

Status: Resolved