Audit Report

wasmd

v1.0

October 17, 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 wasmd Cosmos SDK module.

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/wasmd

Commit hash: `14688c09855ee928a12bcb7cd102a53b78e3cbfb`

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

This audit covers the wasmd Cosmos SDK module. wasmd integrates wasmvm, which allows creation, instantiation, execution, upgrades and other features of CosmWasm smart contracts on Cosmos SDK blockchains.

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 improving 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 in a security audit and vice versa.

Summary of Findings

No	Description	Severity	Status
1	Gzipped wasm binaries with invalid CRC could be used to DOS the chain	Critical	Resolved
2	Contract admins can bypass code ID instantiation permission when migrating contracts	Major	Resolved
3	IBC Querier plugin's unbounded loop could lead to DoS	Major	Partially Resolved
4	Governance permissioned chains are not supporting CosmWasm contracts that dynamically instantiate other contracts	Minor	Resolved
5	When updating a contract's `AccessConfig` subset conditions are not enforced	Minor	Resolved
6	Attribute keys starting with underscores lead to errors, causing smart contract runtime errors	Minor	Acknowledged
7	Updating access configurations can render existing contracts non-compliant	Minor	Acknowledged
8	Possible key collision in `appendToContractHistory` function	Informational	Resolved
9	`autoIncrementID` is a misleading variable name that may impact future maintainability	Informational	Resolved
10	Input label validation can be bypassed using white space, label supports non-printable characters	Informational	Partially Resolved
11	Proposal validations can be improved	Informational	Resolved
12	`handleMigrateProposal` contains unreachable error	Informational	Resolved
13	Typographic and grammar errors found in codebase	Informational	Resolved
14	Lack of event emission when storing code, updating, or clearing an admin through a proposal	Informational	Acknowledged
15	Sudo contract interactions do not consume gas	Informational	Acknowledged

Code Quality Criteria

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

Detailed Findings

1. Gzipped wasm binaries with invalid CRC could be used to DOS the chain

Severity: Critical

In the `create` function in `x/wasm/keeper/keeper.go:181`, the passed `wasmCode` is uncompressed. Gas is not charged during uncompression, but rather further down in line `185` after the error check in line `183`. Consequently, if an error occurs during uncompression, the function returns an error without charging gas for uncompressing the code.

An attacker can consume a high amount of computation and I/O operations without paying for that resource usage by sending a big file to uncompress and causing an error at the last moment. This can be achieved by triggering the uncompression of a valid gzipped file but with an invalid CRC, so the function will read the entire buffer before failing during the CRC check, which happens after uncompressing the whole file.

For the attack, it is important to create a valid gzipped file with a maximum size of `800 kB`. An invalid/bigger file would fail due to the byte prefix check and the `LimitReader` in place. Since the `gasMeter` charges gas proportional to the transaction message size, it is desirable to use a very small gzipped file with an uncompressed size of `800 kB`.

A test case demonstrating the above scenario can be found in appendix 1.

The cost of this attack could further be decreased by sending a `MsgStoreCode` with the code constructed as described above from within a CosmWasm smart contract. Even though `MsgStoreCode` is not directly supported by the default Wasm message encoder `EncodeWasmMsg`, it can be sent using a `StargateMsg`.

Recommendation

We recommend charging gas for the read and decompress operation of the gzipped binary array even in the case of an error.

Status: Resolved

2. Contract admins can bypass code ID instantiation permission when migrating contracts

Severity: Major

The migrate functionality in `x/wasm/keeper/keeper.go:380` does not check whether the caller is authorized to perform a contract migration towards the specified code ID. As a

result, contract admins can bypass any code ID instantiate permission by specifying the restricted code ID when migrating their contract.

A test case demonstrating the above scenario can be found in appendix 2.

Recommendation

We recommend verifying that the caller is authorized to migrate the code ID as seen in `x/wasm/keeper/keeper.go:275-277`.

Status: Resolved

3. IBC Querier plugin's unbounded loop could lead to DoS

Severity: Major

The `IBC Query` plugin is defined in `x/wasm/keeper/query_plugins.go:200`, allowing queries of smart contracts' IBC metadata.

In line `209`, while executing the `ListChannels` query, specific logic is executed to search open channels associated with a given `PortID`. This is achieved by iterating through existing channels using the `IterateChannels` function from `channelKeeper`.

Notably, if `PortID` remains unspecified, all open channels are retrieved.

Consequently, in scenarios where a substantial number of channels are registered, an attacker could exploit this unbounded iteration, leading to a chain-wide DoS.

Recommendation

We recommend incorporating pagination into this query and devising a more optimized approach for querying channels associated with a specific `portID`. Since channels are stored using the `ChannelKey(portID, channelID)` key, direct access to them could be feasible without the need to iterate through all of them.

Status: Partially Resolved

This issue is partially resolved because the fixed version still iterates over all channels inside a specific `portID`. If that `portID` has many channels, an out-of-gas error can still occur.

4. Governance permissioned chains are not supporting CosmWasm contracts that dynamically instantiate other contracts

Severity: Minor

The authz implementation in `x/wasm/keeper/authz_policy.go` does not propagate governance permissions, i. e. a message sent from a CosmWasm smart contract will be

executed in the runtime with an authz object of type `DefaultAuthorizationPolicy` and not `GovAuthorizationPolicy`.

This implies, for example, that contracts cannot instantiate other contracts, which is a common pattern used by teams building CosmWasm smart contracts. Scenario:

- Governance permissioned chain that allows only governance to load code and instantiate a contract:
  `code_upload_access = ACCESS_TYPE_NOBODY`
  `instantiate_default_permission = ACCESS_TYPE_NOBODY`
- Code is uploaded and a contract is instantiated through governance.
- The contract itself cannot dynamically instantiate other contract instances.

Recommendation

We recommend propagating the `GovAuthorizationPolicy` to sub-messages to support the common pattern of contract instantiation from contracts.

An alternative solution would be to add a wasm flag to explicitly say that a contract needs to instantiate other contracts in order to work properly and then evaluate that flag with the provided authz configurations.

Status: Resolved

5. When updating a contract's `AccessConfig` subset conditions are not enforced

Severity: Minor

In the `setAccessConfig` function in `x/wasm/keeper/keeper.go:882`, when updating the `AccessConfig` for a specific contract code, no check is performed on whether the new config is a subset of the global access config as is done in other parts of the codebase. This could lead to inconsistency of the local `AccessConfig` with the global one,

Recommendation

We recommend enforcing the new config to be a subset of the global one, as done in the `keeper.create` function.

Status: Resolved

6. Attribute keys starting with underscores lead to errors, causing smart contract runtime errors

Severity: Minor

The `contractSDKEventAttributes` function in `x/wasm/keeper/events.go:62`, returns an error if an attribute key starts with an underscore _. This will lead to runtime errors for smart contracts 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 runtime error would prevent the emergency withdrawal, putting user funds at risk.

Recommendation

We recommend escaping/adding a prefix to keys starting with an underscore or ignoring them rather than returning an error, and clearly documenting this behavior.

Status: Acknowledged

The client has acknowledged this issue and states that it is a design decision which is well documented in EVENTS.md.

7. Updating access configurations can render existing contracts non-compliant

Severity: Minor

During instantiation, the access config specific to the contract is verified, but the global one is not considered. In fact, the contract's access config is only validated as a subset of the global one during the code creation process in `x/wasm/keeper/keeper.go:176`.

Consequently, if either the upload access config or instantiate access config are modified, existing contracts may no longer comply with the updated configurations.

For instance, changing the instantiation access config from `AccessTypeEverybody` to `AccessTypeOnlyAddress` will not impact previously created codes, which anyone can still instantiate.

Recommendation

We recommend checking not just a contract's access config, but also the global access config upon upload or instantiation.

Status: Acknowledged

The client states that this is a design decision taking into account the need to restrict/relax access for restricted chains and the expectations/user experience of the contract devs and users. The proper way to restrict access globally would be with a chain upgrade that includes

a storage migration. This migration process involves iterating over all codes/instances and applying the desired changes. The upgraded chain should be accompanied by a communication strategy to inform contract developers and users about the implications of the upgrade. Effective communication plays a key role when restricting access.

8. Possible key collision in `appendToContractHistory` function

Severity: Minor

In `x/wasm/keeper/keeper.go:565`, the reverse iteration in the `appendToContractHistory` function may cause a key collision when contract addresses end with zeroes. If contract addresses are not guaranteed to have the same length, the prefix of a zero-ending address may collide with a contact that has a non-zero-ending address.

Additionally, since the store code for `ContractHistory` does not prevent an address collusion in a mixed 20/32 byte address chain, it might cause the transaction to fail when the position counter is not initialized correctly or overflows for a 20-byte address.

Recommendation

We recommend ensuring the counter does not fail from invalid data or overflow in case wasmd is used as a library with custom address generation.

Status: Resolved

9. `autoIncrementID` is a misleading variable name that may impact future maintainability

Severity: Informational

The `autoIncrementID` function in `x/wasm/keeper/keeper.go:956` contains a misleading variable name called `lastIDKey` that may impact future maintainability. The variable name does not accurately reflect its usage – rather than storing the last identifier key, and it is actually used as the next identifier key.

Recommendation

We recommend updating the name of `lastIDKey` to `nextIDKey` in the `autoIncrementID` function in `x/wasm/keeper/keeper.go:956`.

Status: Resolved

10. Input label validation can be bypassed using white space, label supports non-printable characters

Severity: Informational

In `x/wasm/types/validation.go:26-28`, the `validateLabel` function verifies the label is not an empty string without trimming white space from user input. As a result, one can simply provide white space as the contract label, which the validation function does not reject. Additionally, control characters and non-printable characters can be used in the label. This defeats the purpose of the validation and may confuse users.

Recommendation

We recommend trimming all leading and trailing white space before checking whether the user input is an empty string using `strings.TrimSpace(label)` as well as restricting the character set to human-readable characters.

Status: Partially Resolved

The client has added validation that returns an error if the label starts or ends with whitespace. However, control characters and non-printable characters can still be used. The client has opened a GitHub issue to potentially address this issue in the future: https://github.com/CosmWasm/wasmd/issues/1623.

11. Proposal validations can be improved

Severity: Informational

In `x/wasm/types/proposal.go:485-487`, the `ValidateBasic` function for `PinCodesProposal` only checks that the provided `CodeIDs` slice is not empty. Ideally, it should also check that it does not contain duplicated code IDs and that the provided code IDs are not `0`. There is also currently no upper bound on the number of `CodeIDs` in the validation.

This issue is also present in the `ValidateBasic` functions for `UnpinCodesProposal` and `UpdateInstantiateConfigProposal`.

Recommendation

We recommend deduplicating the `CodeIDs` slice and returning an error message if a provided code ID is `0`. We also recommend setting an upper limit to the number of `CodeIDs` in these slices.

Status: Resolved

12. `handleMigrateProposal` contains unreachable error

Severity: Informational

In `x/wasm/keeper/proposal_handler.go:107-112`, there are two `if` statements that return an error if the `err` value is not `nil`. The second `if` statement in line `110` cannot be executed because the first one would always return before it.

Recommendation

We recommend removing the second if statement in lines `110-112`.

Status: Resolved

13. Typographic and grammar errors found in codebase

Severity: Informational

During the audit, several typographical and grammar errors were found that negatively impact readability:

- `proto/cosmwasm/wasm/v1/tx.proto:30`: "Sender is the actor that signed the messages".
- `x/wasm/keeper/api.go:10, 12`: "much"
- `x/wasm/keeper/gas_register.go:59`: "create"
- `x/wasm/keeper/keeper.go:266`: "contract"
- `x/wasm/types/wasmer_engine.go:214`: "probably"
- `x/wasm/types/wasmer_engine.go:143, 157`: "phase"

Recommendation

We recommend correcting these errors as mentioned above.

Status: Resolved

14. Lack of event emission when storing code, updating, or clearing an admin through a proposal

Severity: Informational

In the following cases, events are emitted when actions are performed through messages, but not when they are performed through proposals. This inconsistency might be confusing for users and may lead to issues for off-chain services such as block explorers.

- When storing code through a proposal in `x/wasm/keeper/proposal_handler.go:58`, no event is emitted. This differs from the handler for `StoreCode` messages, which emits an event.

- When updating an admin through a proposal in `x/wasm/keeper/proposal_handler.go:172`, no event is emitted. This differs from the handler for `UpdateAdmin` messages, which emits an event.
- When clearing an admin through a proposal in `x/wasm/keeper/proposal_handler.go:188`, no event is emitted. This differs from the handler for `ClearAdmin` messages, which emits an event.

Recommendation

We recommend emitting events for these actions irrespective of whether they were triggered through messages or proposals.

Status: Acknowledged

The client mentioned that this is solved with gov v1. Gov v1beta1 is deprecated and will be removed soon.

15. Sudo contract interactions do not consume gas

Severity: Informational

The sudo contract interactions do not consume gas throughout the codebase. Although gas is charged through the `tx.GasMeter().ConsumeGas(sudoSetupCosts, "Loading CosmWasm module: sudo")` call in `x/wasm/keeper/keeper.go:461`, sudo proposals eventually go into the `BeginBlocker` or `Endblocker`, where gas is not actually accounted for.

We classify this issue as informational since sudo calls are trusted.

Recommendation

We recommend charging gas for sudo contract interactions.

Status: Acknowledged

The client states that this observation is correct for sudo governance proposals and a design decision by the SDK team for the gov module. There is nothing wasmd can do about this, as gas limits are managed externally. Having said that, calling sudo via a governance proposal is not the only way to execute a sudo callback into the contract. There can be native integrations, as in Tgrade or Osmosis, where the callback is executed on a transaction level.

Appendix

Test case for issue 1

Create a valid 800 kB gzipped file with an invalid CRC:

```
truncate -s 799K test.txt
gzip test.txt
dd if=test.txt.gz bs=1 count=100 of=test.txt.gz
dd if=test.txt.gz bs=1 skip=100 seek=104 of=test.txt.gz
#need to do the modification in the DEFLATE encoded part, see the gzip structure for more info https://en.wikipedia.org/wiki/Gzip#:~:text=%22gzip%22%20is%20often%20also%20used,and%20the%20operating%20system%20ID.
gunzip test.txt.gz #takes time but go into error
```

Then use this test case:

```
func InstantiateHackExampleContract(t testing.TB, ctx sdk.Context, keepers TestKeepers) HackatomExampleInstance {
	contract := StoreExampleContract(t, ctx, keepers, "./testdata/hack.gz")
	verifier, _, verifierAddr := keyPubAddr()
	fundAccounts(t, ctx, keepers.AccountKeeper, keepers.BankKeeper, verifierAddr, contract.InitialAmount)
	beneficiary, _, beneficiaryAddr := keyPubAddr()
	initMsgBz := HackatomExampleInitMsg{
		Verifier:    verifierAddr,
		Beneficiary: beneficiaryAddr,
	}.GetBytes(t)
	initialAmount := sdk.NewCoins(sdk.NewInt64Coin("denom", 100))
	adminAddr := contract.CreatorAddr
	contractAddr, _, err := keepers.ContractKeeper.Instantiate(ctx, contract.CodeID, contract.CreatorAddr, adminAddr, initMsgBz, "demo contract to query", initialAmount)
	require.NoError(t, err)
	return HackatomExampleInstance{
		ExampleContract: contract,
		Contract:        contractAddr,
		Verifier:        verifier,
```

```
            VerifierAddr:    verifierAddr,
            Beneficiary:     beneficiary,
            BeneficiaryAddr: beneficiaryAddr,
      }
}
func TestHack(t *testing.T) {
      mockWasmVM := wasmtesting.MockWasmer{MigrateFn: func(codeID wasmvm.Checksum, env wasmvmtypes.Env, migrateMsg []byte, store wasmvm.KVStore, goapi wasmvm.GoAPI, querier wasmvm.Querier, gasMeter wasmvm.GasMeter, gasLimit uint64, deserCost wasmvmtypes.UFraction) (*wasmvmtypes.Response, uint64, error) {
            return &wasmvmtypes.Response{}, 1, nil
      }}
      wasmtesting.MakeInstantiable(&mockWasmVM)
      ctx, keepers := CreateTestInput(t, false, SupportedFeatures, WithWasmEngine(&mockWasmVM))
      InstantiateHackExampleContract(t, ctx, keepers)
}
```

Test case for issue 2

```
func TestBypassContractInitPermission(t *testing.T) {
	// Test case reproduced in: x/wasm/keeper/keeper_test.go; modified version of `TestMigrateWithDispatchedMessage` test case

	ctx, keepers := CreateTestInput(t, false, SupportedFeatures)
	keeper := keepers.ContractKeeper

	deposit := sdk.NewCoins(sdk.NewInt64Coin("denom", 100000))
	alice := keepers.Faucet.NewFundedAccount(ctx, deposit.Add(deposit...)...)
	bob := keepers.Faucet.NewFundedAccount(ctx, sdk.NewInt64Coin("denom", 100000))

	originalContractID, err := keeper.Create(ctx, bob, hackatomWasm, nil)
	require.NoError(t, err)

	// 1. Alice creates a code ID with `AllowNobody` permission, no one can initialize the code ID
	forbiddenContractID, err := keeper.Create(ctx, alice, hackatomWasm, &types.AllowNobody)
	require.NoError(t, err)

	initMsg := HackatomExampleInitMsg{
		Verifier:    bob,
		Beneficiary: bob,
	}
	initMsgBz := initMsg.GetBytes(t)

	contractAddr, _, err := keepers.ContractKeeper.Instantiate(ctx, originalContractID, bob, bob, initMsgBz, "bob hacker contract", deposit)
	require.NoError(t, err)

	// 2. Bob tries to instantiate the forbidden code ID, an unauthorized error would occur
	_, _, err = keepers.ContractKeeper.Instantiate(ctx, forbiddenContractID, bob, nil, initMsgBz, "bob init forbidden code ID", deposit)
	require.Error(t, sdkerrors.ErrUnauthorized, err)

	// 3. Bob can bypass the `AllowNobody` access permission by performing a contract migration towards the forbidden code ID
	_, err = keeper.Migrate(ctx, contractAddr, bob, forbiddenContractID, initMsgBz)
	require.NoError(t, err)

	// 4. The same goes for `AccessTypeOnlyAddress` access permission, in this example only Alice can instantiate this contract
	onlyCreatorContractID, err := keeper.Create(ctx, alice, hackatomWasm, &types.AccessConfig{Permission: types.AccessTypeOnlyAddress, Address: alice.String()})
	require.NoError(t, err)

	// 5. Bob cannot instantiate code ID as it is only authorized for Alice
```

```
	_, _, err = keepers.ContractKeeper.Instantiate(ctx, onlyCreatorContractID,
bob, nil, initMsgBz, "bob init code ID that is only for alice", deposit)
	require.Error(t, sdkerrors.ErrUnauthorized, err)

	// 6. Bob performs a contract migration to bypass the permissioned
instantiation check
	_, err = keeper.Migrate(ctx, contractAddr, bob, onlyCreatorContractID,
initMsgBz)
	require.NoError(t, err)
}
```