Comparison with Solidity Contracts

First of all, the deploy-execute process consists of 3 steps rather than 2. While Ethereum was built around the concept of many unique contracts, each possibly custom-made for any bilateral agreement, the reality seems to show that writing a bug-free contract is harder than originally thought, and a majority are copies of standard templates like OpenZepellin. With that in mind and conscious of the overhead of uploading and validating wasm code, we define the following 3 phases of a contract:

- Upload Code - Upload some optimized wasm code, no state nor contract address (example Standard ERC20 contract)
- Instantiate Contract - Instantiate a code reference with some initial state, creates a new address (for example set token
- name, max issuance, etc formy
- ERC20 token)
- Execute Contract - This may support many different calls, but they are all unprivileged usage of a previously
- instantiated contract depends on the contract design (example: Send ERC20 token, grant approval to another contract)

Just like Ethereum, contract instantiation and execution are metered and require gas. Furthermore, both instantiation and execution allow the signer to send some tokens to the contract along with the message. Two key differences are that sending tokens directly to a contract, eg. viaSendMsg , while possible,does not trigger any contract code . This is a clear design decision to reduce possible attack vectors. It doesn't make anything impossible but requires all execution of the contract to beexplicitly requested .

Avoiding Reentrancy Attacks

Another big difference is that we avoid all reentrancy attacks by design. This point deserves an article by itself, but in short a large class of exploits in Ethereum is based on this trick . The idea is that in the middle of the execution of a function on Contract A, it calls a second contract (explicitly or implicitly via send). This transfers control to contract B, which can now execute code, and call into Contract A again. Now there are two copies of Contract A running, and unless you are very, very careful about managing the state before executing any remote contract or make very strict gas limits in sub-calls, this can trigger undefined behavior in Contract A and a clever hacker can reentrancy this as a basis for exploits, such as the DAO hack.

Cosmwasm avoids this completely by preventing any contract from calling another one directly. Clearly, we want to allow composition, but inline function calls to malicious code create a security nightmare. The approach taken with CosmWasm is to allow any contract toreturn a list of messagesto be executed in the same transaction . This means that a contract can request a send to happen after it is finished (eg. release escrow), or call into another contract. If the future messages fail, then the entire transaction reverts, including updates to the contract's state. This allows for atomic composition and quite a few security guarantees, with the only real downside that you cannot view the results of executing another contract, rather you can just do "revert on error".

Sometimes we will need information from another contract, and with the 0.8 release, we added synchronous queries to other contracts or underlying Cosmos SDK modules. These Queries only have access to a read-only database snapshot and be unable to modify state or send messages to other modules, thus avoiding any possible reentrancy concerns.

Resource Limits

Beyond exploits (such as the reentrancy attack), another attack vector for smart contracts is a denial of service attacks. A malicious actor could upload a contract that ran an infinite loop to halt the chain or write tons of data to fill up the disk. Web Assembly provides a tight sandbox with no default access to the OS, so we only need to worry about providing tight resource limits for the smart contracts. All developers should be aware of these limits.

Memory Usage - When instantiating a Wasm VM, it is provided 32MB of RAM by default. This is to store the byte code as well as all memory used by running the process (stack and heap). This should be plenty large for almost any contract, but maybe some complex zero-knowledge circuits would hit limits there. It is also small enough to ensure that contracts have minimal impact on memory usage of the blockchain.

CPU Usage - TheWasmer Runtime that we use, can inject metering logic into the wasm code. It calculates prices for various operations and charges and checks limits before every jump statement (loop, function call, etc), to produce a deterministic gas price regardless of CPU speed, platform, etc. Before executing a contract, a wasm gas limit is set based on the remaining Cosmos SDK gas, and gas is deducted at the end of the contract (there is a constant multiplier to convert, currently 100 wasm gas to 1 SDK gas). This puts a hard limit on any CPU computations as you must pay for the cycles used.

Disk Usage - All disk access is via reads and writes on the KVStore. The Cosmos SDK already enforces gas payments for KVStore access . Since all disk access in the contracts is made via callbacks into the SDK, this is charged there. If one were to integrate CosmWasm in another runtime, you would have to make sure to charge for access there as well.

Lessons Learned from Ethereum

Ethereum is the grandfather of all blockchain smart contract platforms and has far more usage and real-world experience than any other platform. We cannot discount this knowledge but instead learn from their successes and failures to produce a more robust smart contract platform.

They have compiled a list ofall known Ethereum attack vectors along with mitigation strategies. We shall compare Cosmwasm against this list to see how much of this applies here. Many of these attack vectors are closed by design. A number remain and a section is planned on avoiding the remaining such issues.

✔Reentrancy

In cosmwasm, we return messages to execute other contracts, in the same atomic operation, butafter the contract has finished. This is based on the actor model and avoids the possibility of reentrancy attacks - there is never a volatile state when a contract is called.

✔Arithmetic under/overflows

Rust allows you to simply setoverflow-checks = true in theCargo manifest to abort the program if any overflow is detected. No way to opt out of safe math.

⚠Unexpected Ether

Bad design pattern

This involves a contract depending on complete control of its balance. A design pattern that should be avoided in any contract system. In CosmWasm, contracts are not called when tokens are sent to them, but they can query their current balance when they are called. You can note that thesample escrow contract doesn't record how much was sent to it during initialization, but ratherreleases the current balance when paying out or refunding the amount. This ensures no tokens get stuck.

✔Delegate Call

We don't have such Delegate Call logic in CosmWasm. You can import modules, but they are linked together at compile time, which allows them to be tested as a whole, and no subtle entry points inside of a contract's logic.

✔Default Visibilities

Rather than auto-generating entry points for every function/method in your code (and worse yet, assuming public if not specified), the developer must clearly define a list of messages to be handled and dispatch them to the proper functions. It is impossible to accidentally expose a function this way.

⚠Entropy Illusion

Planned Fix

The block hashes (and last digits of timestamps) are even more easily manipulated by block proposers in Tendermint, than with miners in Ethereum. They should not be used for randomness. There is work planned to provide a secure random beacon and expose this secure source of entropy to smart contracts.

✔External Contract Referencing

Planned Mitigation

If you call a contract with a givenExecuteMsg , this just requires the contract to have the specified API but says nothing of

the code there. I could upload malicious code with the same API as the desired contract (or a superset of the API), and ask you to call it - either directly or from a contract. This can be used to steal funds and in fact wedemo this in the tutorial.

There are two mitigations here. The first is that in CosmWasm, you don't need to call out to solidity libraries at runtime to deal with size limits, but are encouraged to link all the needed code into one wasm blob. This alone removes most usage of the external contract references.

The other mitigation is allowing users to quickly find verified rust source behind the wasm contract on the chain. This approach isused by etherscan , where developers can publish the source code, and it will compile the code. If the same bytecode is on the chain, we know can prove it came from this rust source. We have built the deterministic build system for rust wasm, and havesimple tooling to validate the original source code. We alsoreleased a code explorer that allows you to browse contracts and locally verify the source code in one command.

✔Short Address/Parameter Attack

This is an exploit that takes advantage of the RLP encoding mechanism and fixed 32-byte stack size. It does not apply to our type-checking JSON parser.

✔Unchecked CALL Return Values

CosmWasm does not allow calling other contracts directly, but rather sent messages will be dispatched by a router. The router will check the result of all messages, and ifany message in the chain returns an error, the entire transaction is aborted, and state changes rolled back. This allows you to safely focus on the success case when scheduling calls to other contracts, knowing all states will be rolled back if it does not go as planned.

⚠Race Conditions/Front Running

This is a general problem with all blockchains. The signed message is gossiped among all nodes before a block is formed. A key miner/validator could create another transaction and insert it before yours (maybe delaying yours). This is often not an issue but shows up quite a bit in distributed exchanges. If there is a standing sell order at 90 and I place a buy order at 100, it would normally just match at 90. However, a miner could insert two transactions between, one to buy at 90, then the other to sell at 100, then delay your transaction to the end. You would end up accepting their offer at 100 and they would make a profit of 10 tokens just for doing arbitrage.

This is also an issue in high-frequency trading and any system that relies on ordering between clients to determine the outcome, just more pronounced in blockchain as the delays are on the order of seconds, not microseconds. For most applications, this is not an issue, and for decentralized exchanges, there are designs with eg. batch auctions that eliminate the potential of front running.

⚠Denial of Service

limited circumstances

The idea is that if the contract relies on some external user-defined input, it could be set up in a way that it would run out of gas processing it. Many of the cases there should not affect CosmWasm, especially as wasm runs much faster and CPU gas limits allow huge amounts of processing in one transaction (including ed25519 signature verification in wasm without a precompile). However, looping over a user-controlled number of keys in the storage will run out of gas quickly.

✔Block Timestamp Manipulation

Tendermint providesBFT Timestamps in all the blockchain headers. This means that you need a majority of the validators to collude to manipulate the timestamp, and it can be as trusted as the blockchain itself. (That same majority could halt the chain or work on a fork)

✔Constructors with Care

This is an idiosyncrasy of the solidity language with constructor naming. It is highly unlikely you would ever renameinit in

cosmwasm, and if you did, it would fail to compile rather than produce a backdoor.

✔Uninitialised Storage Pointers

CosmWasm doesn't automatically fill in variables, you must explicitly load them from storage. And rust does not allow uninitialized variables anywhere (unless you start writingunsafe blocks, and make a special call to access uninitialized memory... but then you are asking for trouble). Making storage explicit rather than implicit removes this class of failures.

✔Floating Points and Precision

Both Solidity and CosmWasm have no support for floating-point operations, due to possible non-determinism in rounding (which is CPU dependent). Solidity has no alternative to doing integer math and many devs hand-roll integer approximations to decimal numbers, which may introduce rounding errors.

In CosmWasm, You can import any rust package, and simply pick an appropriate package and use it internally. Likerust_decimal , "a Decimal implementation written in pure Rust suitable for financial calculations that require significant integral and fractional digits with no round-off errors.". Orfixed to provide fixed-point decimal math. It supports up to 128-bit numbers, which is enough for 18 digits before the decimal and 18 afterward, which should be enough for any use case.

✔Tx.Origin Authentication

CosmWasm doesn't exposetx.origin , but only the contract or user directly calling the contract asparams.message.signer . This means it is impossible to rely on the wrong authentication, as there is only one value to compare. Previous Contract Composition Next Contract Semantics * Avoiding Reentrancy Attacks * Resource Limits * Lessons Learned from Ethereum * * ✔ Reentrancy * * ✔ Arithmetic under/overflows * * ⚠ Unexpected Ether * * ✔ Delegate Call * * ✔ Default Visibilities * * ⚠ Entropy Illusion * * ✔ External Contract Referencing * * ✔ Short Address/Parameter Attack * * ✔ Unchecked CALL Return Values * * ⚠ Race Conditions/Front Running * * ⚠ Denial of Service * * ✔ Block Timestamp Manipulation * * ✔ Constructors with Care * * ✔ Uninitialised Storage Pointers * * ✔ Floating Points and Precision * * ✔ Tx.Origin Authentication