Manual - < v1.11

Since the V1.11 Upgrade Secret now supports native migrateability for contracts The Secret Network's version of the CosmWasm runtime disables the ability to natively migrate contracts. Data written to storage using some contract can only ever be accessed by that same contract. This is done for trust and security reasons. Once a user sends some information to a secret contract, there should not be an easy way for anyone to change the code that has access to this information. If it was possible, then the new code would be able to leak any information entrusted by the user to the old code.

That being said, contracts are fully programmable, and "not easy" is not impossible. If it was, we wouldn't even be here 😉. In some cases, for example when wanting to deliver support to users and deploy bug fixes, it makes sense to have the ability to change the code. This is especially true for services which wish to change algorithms, add features, etc. For these use cases, contracts can be written to allow various modes of migration. Doing so means that the anyone inspecting the source code will be able toknow that such a migration is possible and will require that users have a higher level of trust in the contract administrators, as they effectively install a backdoor in their product.

The contracts in this repository show various migration strategies that can be used and adapted in real-world contract. These should not be considered production-ready examples, and some applications may have more specific or creative migration paths.

For examples of implementations of the techniques described here see:https://github.com/scrtlabs/secret-migrate-example

Did you know? Contract migration/upgradability in a privacy-preserving way is on the roadmap for Secret Network

Handoff

The idea of a handoff architecture is that the original deployed contract is written in a way that allows a contract admin to give a new contract permission to directly collect private information from it. In order to add a basic level of protection, the reference in this repository makes sure that the information can only be collected by a contract, and not just a regular user.

This is done by exchanging a secret with a contract through a callback. The admin would provide the first contract with the address and hash of the new contract. The first contract would then generate a secret based on some source entropy (internal or external) which it would send to the second contract (A malicious admin can still write a contract that would leek this secret, but we already decided we have to trust the admin). The second contract can then query the first contract, either once or with some paging, to retrieve and store all the previously hidden information. An important detail shown in the example is that the first contract essentially gets locked as soon as the migration starts. This is done in order to make sure the data stored in it doesn't change anymore. new interactions should be with the new contract, after the admin completes the migration.

Facade

In theory, one could write a contract that only forwards messages to another contract that is configured by the contract admin. Users would then only interact with the facade, rather than the implementation contracts. You can imagine that it would define a small, reserved interface of messages that do not get forwarded, and would only be seen by the facade. Messages such asChangeOwner ,ChangeImplemntation , etc. This would allow great flexibility for contract authors to just deploy a new version of a contract, and point the facade to the new implementation. Paired with theHandoff technique, you could have a complete change of implementation, with only a short downtime. (The facade contract can even be written in a way that informs users that the service is temporarily down, and that they should retry in a couple minutes)

Unfortunately, this is currently not feasible, at least not in a pretty way, because of limitations of the available json parsing libraries. The only one i'm aware of that can run on an environment that forbids floats isserde-json-wasm , but it doesn't support parsing unstructured messages (like e.g.,serde_json::Value). A workaround then, is to include the message to the "backend" contract as a serialized JSON in a string field of the message to the facade. It's not as pretty but can be implemented without having to start rewriting ecosystem packages.

Another, less flexible alternative, is to define the full API that you will commit to for the rest of the application's lifetime. Then. you can just parse-reserialize messages that aren't meant for the facade itself.

Configurable Factory

This is more of a migration technique for products rather than specific contracts, but it's worth mentioning.

The idea is that if your application needs to deploy a new contract every time it serves customers (e.g., game rooms, bids, etc.), then you can write a factory contract that handles some things for you. The factory contract can be used for many things, from providing one authentication method for queries to multiple contracts, for aggregating usage statistics on-chain, for sharing information between contracts in a structured way, and more. Another use for it, is changing the implementation ofnew contract that will be deployed in the future.

You can think of the factory as a sort of facade, so it can also include a handle that would allow the admin to reconfigure the code-id, and maybe even init parameters, used to open new contracts (game rooms, bids, etc.).

An example of this technique being used in production isBaedrik's Secret Auctions .

SNIP-20 swap

A SNIP-20 contract can also implement theReceive interface and allow users of one token toSend a swap request to the new token. The new token will lock those funds in its custody for the rest of time, and mint new tokens, with some exchange rate, to the sender address. This technique will require zero preparation by the original token contract.

A similar technique can be used by applications with similar ideas to SNIP-20.

Best practices

Admin keys

In case of emergencies, it can be useful to have certain admin functions to alter the functionality of the contract in some way that are only usable by the admin. For example, if your contract receives funds and forwards them to a different address, but you lose the keys to the to that address, you want to ensure their is a function implemented that will enable you to change it. Or possibly there may be some situation where you want to deactivate the contract (aside from the admin function to reactivate it).

It's important to consider what possibly go wrong with the contract and what can be done to fix it ahead of time, and implement those fixes. On the flip side, making too many functions alterable by the admin may make your dapp more centralized and users may become more cautious to use it. Balance these two sides carefully during dapp creation.

Data Migration

In some cases, you may want to write a new updated smart contract. This can be for multiple reasons, such as fixing bugs or adding new features. However, because you cannot just change the code of a contract already uploaded on the chain, the upgrade process becomes significantly more complex than web2 and there are many things you will need to consider.

Block size limitations

Due to the expense of sending large amounts of data, it would be unreasonable or impossible to send all data to a new contract in a single transaction if you store a significant amount, such as in the case of a token contract which may have tens of thousands of users. You will have to set up functions to transfer data strategically and in manageable chunks.

Decentralize by making users their own admin

If a single admin calls a function that would transfer over all user data to a new contract, in many cases that would also mean they could simply read the data themselves by transferring it to a contract they have total read access to. To prevent this risk to user privacy, it is wise to only give your users permission to transfer over their own data. Using the example of a token contract once again, allow only the users permission to transfer their token balance to the upgraded token contract. Otherwise there is a risk of an admin reading their balance without their permission.

Data removal

In some cases, you may want to ensure once the data is transferred it is immediately deleted, or otherwise prevented from being accessed again from the old contract. This is to prevent the possibility of a user taking advantage of their data existing on two contracts simultaneously. For example, if an NFT contract was updated and the NFTs were moved over to the new contract, but the old contract did not remove them after transferring, you could risk your users selling both the old and new copy their NFTs. Your specific contract may not require data deletion like this, but it is important to consider all possibilities and think from the perspective of potential attackers to decide if this is a risk you need to prevent.

Last updated6 months ago On this page

Was this helpful? Edit on GitHub Export as PDF