Mitigate privacy risks - full guide

Full example guide on mitigating privacy risks.

Execute messages

Here we use a sample contract for selling a secret message to illustrate padding, gas evaporation, and side-chain attack prevention using a trusted actor to confirm transactions. In the contract we define three methods: posting a secret for sale, buying a secret, and confirming purchases. All three message types have two optional parameters:gas_target andpadding . These will be used to help mask which message is being called.

```

Copy

# [derive(Serialize,Deserialize,Clone,Debug,Eq,PartialEq,JsonSchema)]

# [serde(rename_all="snake_case")]

pubenumExecuteMsg{ PostSecret{ /// secret message message:String, /// price to purchase secret price:Uint128, /// optional gas target for evaporation gas_target:Option, /// optional message length padding padding:Option, }, BuySecret{ /// unique id for message message_id:String, /// optional gas target for evaporation gas_target:u32, /// optional message length padding padding:Option, }, ConfirmPurchase{ /// transaction id being confirmed tx_id:String, /// optional gas target for evaporation gas_target:u32, /// optional message length padding padding:Option, } }

```

Using padding and evaporation inexecute

Padding and evaporation are parameters that are set for all message types.

Thepadding parameter is used by the client to pad the size of message being sent to the contract. Thepadding parameter is not used within the contract itself and is only necessary when using Keplr's amino format. In other cases the dapp or wallet can simply add extra space characters to the end of the json message.

We also want to pad the result sent back to the client. We can use thepad_handle_result function from secret toolkit to format the response to a fixed block size. If you have responses that are large, you will need a sufficiently large block size to successfully mask the method being run.

After padding the result at the end ofexecute entry point function, we usegas_target along with the evaporation api functions to evaporate any remaining gas to hit the gas target. Note, we simply ignore both thepadding and thegas_target message parameters using.. in the match statement.

```

Copy

# [entry_point]

pubfnexecute(deps:DepsMut, env:Env, info:MessageInfo, msg:ExecuteMsg)->StdResult{ letresponse=matchmsg { ExecuteMsg::PostSecret{ message, price,..}=>try_post_secret(...), ExecuteMsg::BuySecret{ message_id,..}=>try_buy_secret(...), ExecuteMsg::ConfirmPurchase{ tx_id,..}=>try_confirm_purchase(...), };

// use secret toolkit pad_handle_result to pad the response to blocks of 256 bytes letresponse=pad_handle_result(response,256);

// evaporate any remaining gas ifletSome(gas_target)=msg.gas_target { letgas_used=deps.api.check_gas()?asu32; ifgas_used<gas_target { letto_evaporate=gas_target-gas_used; deps.api.gas_evaporate(to_evaporate)?; } }

// return the response response }

```

Making data written to chain constant size

Just like we want to use padding to hide the message type when sending between the client and the contract, the number of bytes written to the chain can leak information about what is written. For example, a coin amount stored as aUint128 will write a different number of bytes depending on the size of the number, and also in our example the secret message could leak information based on its length. For data structs we want to store on chain we usually create astored version and create conversion methods between the unstored and stored versions. Note in the example, we use the secret toolkitspace_pad function to make all secrets stored as 256 bytes long masking the content.

```

Copy

# [derive(Serialize,Deserialize,Clone,Debug,PartialEq,Eq,JsonSchema)]

# [serde(rename_all="snake_case")]

pubstructSecret{ pubowner:Addr, pubmessage:String, pubprice:Uint128, }

implSecret{ pubfnto_stored(self, api:&dynApi)->StdResult{ letmutmessage=self.message.as_bytes().to_vec(); space_pad(&mutmessage,256);
```

Ok(StoredSecret{ owner:api.addr_canonicalize(self.owner.as_str())?, message, price:self.price.u128(), }) } }

[derive(Serialize,Deserialize,Clone,Debug,PartialEq,Eq,JsonSchema)]

[serde(rename_all="snake_case")]

pubstructStoredSecret{ pubowner:CanonicalAddr, pubmessage:Vec, pubprice:u128, }

```

Preventing a side chain attack

Padding and evaporation help to protect privacy about what types of functions are being called in a contract and about what data is being read and written from storage. However, our contract is still vulnerable to a side chain attack if a buyer can simply purchase a secret without a trusted actor confirming the transaction. This is because a malicious actor could fork the chain and buy the secret on the forked chain without ever paying any coins on the mainnet.

In our case a trusted actor who can confirm the purchase is the seller of the secret. If all purchases must have theConfirmPurchase transaction performed after buying, then it is impossible for the side-chain attack to occur. Other contracts that need to implement this functionality might require a trusted third-party actor instead, for example a game contract where turns are executed on-chain.

Last updated3 months ago On this page *Execute messages * Using padding and evaporation in execute * Making data written to chain constant size * Preventing a side chain attack

Was this helpful? Edit on GitHub Export as PDF
```