// HALBORN

Archway - Migration Contract

CosmWasm Smart Contract Security Audit

Prepared by: **Halborn**

Date of Engagement: **March 6th, 2023 - March 10th, 2023**

Visit: **Halborn.com**

DOCUMENT REVISION HISTORY

VERSION	MODIFICATION	DATE	AUTHOR
0.1	Document Creation	03/06/2023	Luis Quispe Gonzales
0.2	Document Update	03/08/2023	Luis Quispe Gonzales
0.3	Draft Version	03/13/2023	Luis Quispe Gonzales
0.4	Draft Review	03/14/2023	Gabi Urrutia
1.0	Remediation Plan	03/16/2023	Luis Quispe Gonzales
1.1	Remediation Plan Review	03/16/2023	Gabi Urrutia

CONTACTS

CONTACT	COMPANY	EMAIL
Rob Behnke	Halborn	Rob.Behnke@halborn.com
Steven Walbroehl	Halborn	Steven.Walbroehl@halborn.com
Gabi Urrutia	Halborn	Gabi.Urrutia@halborn.com
Luis Quispe Gonzales	Halborn	Luis.QuispeGonzales@halborn.com

EXECUTIVE OVERVIEW

1.1 INTRODUCTION

Archway engaged Halborn to conduct a security audit on their smart contracts beginning on March 6th, 2023 and ending on March 10th, 2023. The security assessment was scoped to the smart contracts provided to the Halborn team.

1.2 AUDIT SUMMARY

The team at Halborn was provided one week for the engagement and assigned a full-time security engineer to audit the security of the smart contract. The security engineer is a blockchain and smart-contract security expert with advanced penetration testing, smart-contract hacking, and deep knowledge of multiple blockchain protocols.

The purpose of this audit is to:

- Ensure that smart contract functions operate as intended
- Identify potential security issues with the smart contracts

In summary, Halborn identified some improvements to reduce the likelihood and impact of risks, which were mostly addressed by the Archway team. The main ones are the following:

- Replace the '&&' operator for a '||' in the conditional of the execute_deploy function.
- Rely on indexes of the list of owners for handling voting or restrict ownership transfers if a proposal is active.
- Verify that the list of owners doesn't have duplicate entries and is not empty before further execution.
- Split ownership transfer functionality to allow the recipient to complete the transfer.

1.3 TEST APPROACH & METHODOLOGY

Halborn performed a combination of manual review of the code and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of the smart contract audit. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance coverage of smart contracts and can quickly identify items that do not follow security best practices. The following phases and associated tools were used throughout the term of the audit:

- Research into architecture, purpose, and use of the platform.
- Manual code read and walkthrough.
- Manual assessment of use and safety for the critical Rust variables and functions in scope to identify any contracts logic related vulnerability.
- Fuzz testing (Halborn custom fuzzing tool)
- Checking the test coverage (cargo tarpaulin)
- Scanning of Rust files for vulnerabilities (cargo audit)

RISK METHODOLOGY:

Vulnerabilities or issues observed by Halborn are ranked based on the risk assessment methodology by measuring the **LIKELIHOOD** of a security incident and the **IMPACT** should an incident occur. This framework works for communicating the characteristics and impacts of technology vulnerabilities. The quantitative model ensures repeatable and accurate measurement while enabling users to see the underlying vulnerability characteristics that were used to generate the Risk scores. For every vulnerability, a risk level will be calculated on a scale of 5 to 1 with 5 being the highest likelihood or impact.

RISK SCALE - LIKELIHOOD

5 - Almost certain an incident will occur.
4 - High probability of an incident occurring.

3 - Potential of a security incident in the long term.
2 - Low probability of an incident occurring.
1 - Very unlikely issue will cause an incident.

RISK SCALE - IMPACT

5 - May cause devastating and unrecoverable impact or loss.
4 - May cause a significant level of impact or loss.
3 - May cause a partial impact or loss to many.
2 - May cause temporary impact or loss.
1 - May cause minimal or un-noticeable impact.

The risk level is then calculated using a sum of these two values, creating a value of 10 to 1 with 10 being the highest level of security risk.

10 - CRITICAL
9 - 8 - HIGH
7 - 6 - MEDIUM
5 - 4 - LOW
3 - 1 - VERY LOW AND INFORMATIONAL

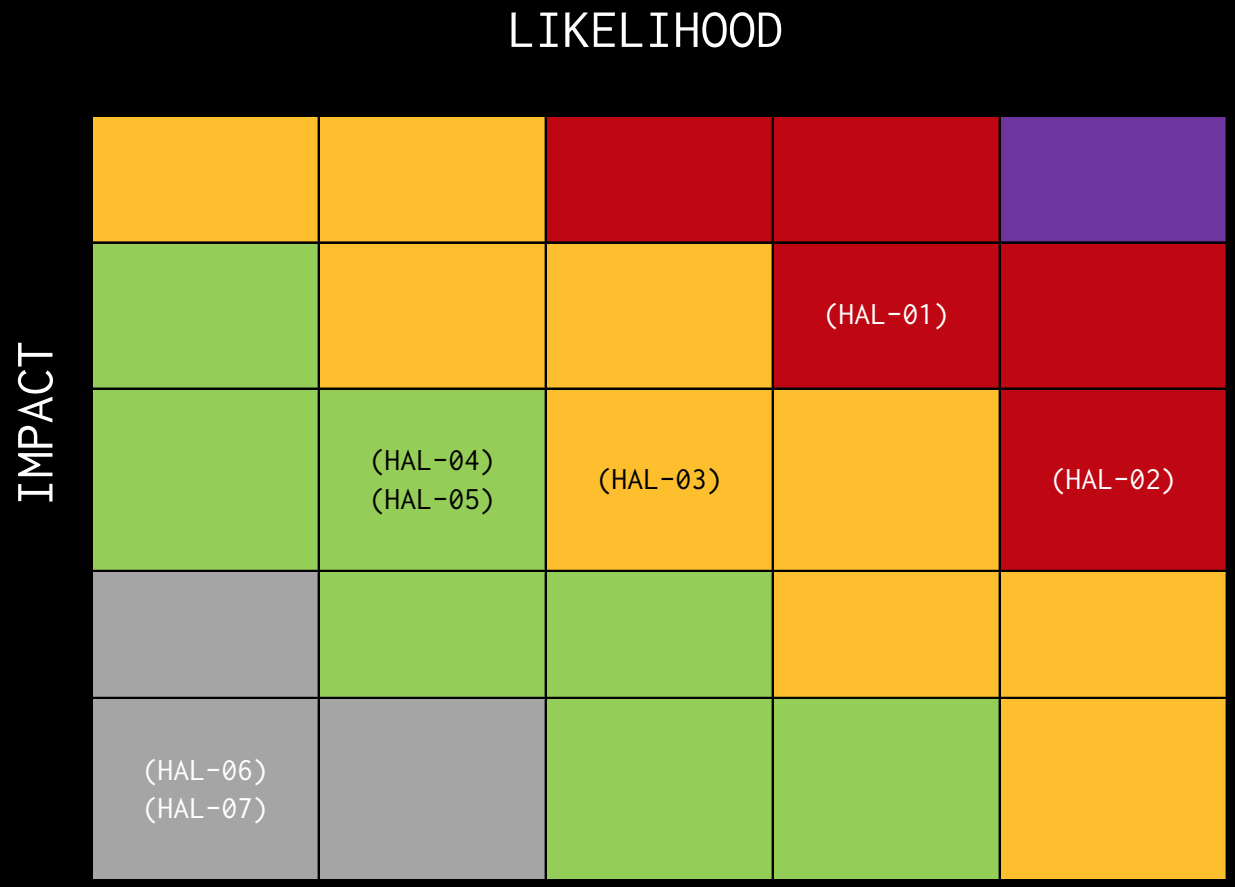
1.4 SCOPE

1. CosmWasm Smart Contracts
   (a) Repository: vesting-contracts
   (b) Commit ID: e4e4836
   (c) Contract in scope:
      - deployer

Out-of-scope: External libraries and financial related attacks

2. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	2	1	2	2



EXECUTIVE OVERVIEW

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
(HAL-01) POSSIBILITY TO NEVER REACH QUORUM ON SOME VESTING CONTRACTS DEPLOYED	High	SOLVED - 03/15/2023
(HAL-02) UNEXPECTED RESULTS FOR PROPOSALS VOTING WHEN TRANSFERRING OWNERSHIP	High	SOLVED - 03/16/2023
(HAL-03) LIST OF OWNERS COULD HAVE DUPLICATE ENTRIES	Medium	SOLVED - 03/15/2023
(HAL-04) OWNERSHIP CAN BE TRANSFERRED WITHOUT CONFIRMATION	Low	RISK ACCEPTED
(HAL-05) LIST OF OWNERS COULD BE EMPTY	Low	SOLVED - 03/16/2023
(HAL-06) USING COUNTERS INSTEAD OF LISTS COULD SAVE SOME GAS ON PROPOSAL VOTING	Informational	ACKNOWLEDGED
(HAL-07) UNIMPLEMENTED FUNCTION FOR PROPOSALS QUERYING	Informational	FUTURE RELEASE

FINDINGS & TECH DETAILS

3.1 (HAL-01) POSSIBILITY TO NEVER REACH QUORUM ON SOME VESTING CONTRACTS DEPLOYED - HIGH

Description:

`execute_deploy` function in **deploy** contract tries to verify that the value of quorum is within a valid range (0% - 100%), but the conditional is using an `&&` operator instead of a `||`. As consequence, this function allows to deploy vesting contracts which quorum values are greater than 100%.

If a quorum is (mistakenly) set to a value greater than 100%, the quorum will never be reached on any of the proposals during the voting, so no one will be able to handle the migration process of the affected contracts in future.

It is worth noting that this situation could affect different vesting contracts due to the root issue is located in the function used to deploy them. Here is a proof of concept showing how to exploit this security issue:

Proof of Concept:

1. A vesting contract is deployed and the quorum is mistakenly set to **1000%**, instead of **100%**.

```
fn mistaken_deploy(deps: DepsMut) {
    CONTRACT_GOVERNANCE.Map<String, GovernanceSettings>
        .save(
            store: deps.storage,
            k: CONTRACT_ADDR.to_string(),
            data: &GovernanceSettings {
                quorum: Decimal::raw(1_000_000_000_000_000_0000u128), // Mistake: 1000% instead of 100%
                admins: vec![ALICE.to_string(), BOB.to_string(), CHARLIE.to_string()],
            },
        ) Result<(), StdError>
        .unwrap();
}
```

```rust
#[test]
▶ Run Test | Debug
fn voting_not_reached_quorum() {
    let mut deps: OwnedDeps<MemoryStorage, …> = mock_dependencies();
    let env: Env = mock_env();

    init(deps: deps.as_mut());
    mistaken_deploy(deps: deps.as_mut());
```

2. One of the owners creates a proposal to update the contract admin.

```rust
// we create a proposal from Charlie to update the contract admin.
let _r: Response = execute_new_proposal(
    deps: deps.as_mut(),
    env.clone(),
    info: mock_info(sender: CHARLIE, funds: &[]),
    contract_addr: CONTRACT_ADDR.to_string(),
    proposal: ProposalMsg::UpdateAdmin {
        admin: "new".to_string(),
    },
) Result<Response, ContractError>
.expect(msg: "unexpected error");
```

3. All owners vote in favor of the proposal.

```rust
// vote Alice accept
vote(deps: deps.as_mut(), &env, who: ALICE, vote: VoteOption::Accept).expect(msg: "unexpected error");

// vote Bob accept
vote(deps: deps.as_mut(), &env, who: BOB, vote: VoteOption::Accept).expect(msg: "unexpected error");

// vote Charlie accept
vote(deps: deps.as_mut(), &env, who: CHARLIE, vote: VoteOption::Accept).expect(msg: "unexpected error");
```

4. Despite the voting result, the quorum will never be reached, i.e.: proposal status will remain as InProgress and won't be executed.

```rust
// Status of proposal
let status: ProposalStatus = PROPOSALS Map<(String, u64), Proposal>
.load(store: deps.as_ref().storage, k: (CONTRACT_ADDR.to_string(), 0)) Result<Proposal, StdError>
.unwrap().status;

assert_eq!(status, ProposalStatus::InProgress);
```

```
    Finished test [unoptimized + debuginfo] target(s) in 1.40s
     Running unittests src/lib.rs (target/debug/deps/archway_vesting_deployer-417295f7f6ffda70)

running 1 test
test contract::tests::voting_not_reached_quorum ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 10 filtered out; finished in 0.00s
```

Code Location:

Listing 1: contracts/deployer/src/contract.rs (Line 170)

```
170 if quorum > Decimal::percent(100) && quorum < Decimal::percent(0)
 ↳ {
171     return Err(ContractError::BadRequest(
172         "invalid quorum value, must be a percentage".to_string(),
173     ));
174 }
175
176 TRANSIENT_CONTRACT_GOVERNANCE.save(deps.storage, &
 ↳ GovernanceSettings { quorum, admins })?;
```

Risk Level:

Likelihood - 4
Impact - 4

Recommendation:

Replace the `&&` operator for a `||` in the conditional shown above for the `execute_deploy` function.

Remediation plan:

SOLVED: The issue was fixed in commit `853310f`.

3.2 (HAL-02) UNEXPECTED RESULTS FOR PROPOSALS VOTING WHEN TRANSFERRING OWNERSHIP - HIGH

Description:

Each vesting contract deployed has its own list of owners, which can create proposals and vote accordingly to handle the migration process of the contract. If one of the owners calls `execute_transfer_admin` function after some proposals have been posted (and before they expire), the following unexpected results can occur:

- The new user won't be able to vote on any of the proposals despite being an owner because his new address is not registered on the proposals. In some edge scenarios, this issue could create a temporary denial-of-service for the quorum mechanism.
- A user who belonged to a list of owners when some proposals were posted (but not anymore), will continue to be able to vote on any of them despite not being an owner.

Proof of Concept:

1. One of the owners creates a proposal to update the contract admin.

```
fn voting_after_ownership_transfer() {
    let mut deps: OwnedDeps<MemoryStorage, …> = mock_dependencies();
    let env: Env = mock_env();

    init(deps: deps.as_mut());
    deploy(deps: deps.as_mut());

    // we create a proposal from Alice to update the contract admin.
    let _r: Response = execute_new_proposal(
        deps: deps.as_mut(),
        env.clone(),
        info: mock_info(sender: ALICE, funds: &[]),
        contract_addr: CONTRACT_ADDR.to_string(),
        proposal: ProposalMsg::UpdateAdmin {
            admin: "new".to_string(),
        },
    ) Result<Response, ContractError>
    .expect(msg: "unexpected error");
```

2. Bob transfers his ownership to a new address: **NEW_BOB**.

```
execute_transfer_admin(
    deps: deps.as_mut(),
    env.clone(),
    info: mock_info(sender: BOB, funds: &[]),
    contract_addr: CONTRACT_ADDR.to_string(),
    new_admin: NEW_BOB.to_string(),
) Result<Response, ContractError>
.expect(msg: "unexpected error");
```

3. When Bob tries to vote with his new address, the operation will panic because **NEW_BOB** is not registered as an owner in the proposal, despite being a real owner.

```
// Alice votes
vote(deps: deps.as_mut(), &env, who: ALICE, vote: VoteOption::Accept).expect(msg: "unexpected error");

// Bob's new address (NEW_BOB) tries to vote
vote(deps: deps.as_mut(), &env, who: NEW_BOB, vote: VoteOption::Accept).expect(msg: "unexpected error");
```

```
    Finished test [unoptimized + debuginfo] target(s) in 0.06s
     Running unittests src/lib.rs (target/debug/deps/archway_vesting_deployer-417295f7f6ffda70)

running 1 test
thread 'contract::tests::voting_after_ownership_transfer' panicked at 'unexpected error: Unauthorized("new_bob not an admin")', cont
racts/deployer/src/contract.rs:498:64
```

Code Location:

execute_new_proposal function stores the **current** list of owners to **CONTRACT_GOVERNANCE** when a new proposal is created:

Listing 2: contracts/deployer/src/contract.rs (Lines 117,120)

```
109 fn execute_new_proposal(
110   deps: DepsMut,
111   env: Env,
112   info: MessageInfo,
113   contract_addr: String,
114   proposal: ProposalMsg,
115 ) -> Result<Response, ContractError> {
116   let conf = CONFIG.load(deps.storage).expect("config must load");
 ↳  // TODO maybe expiration time can be decided by the contract gov
 ↳ settings
117   let settings = CONTRACT_GOVERNANCE.load(deps.storage,
 ↳ contract_addr.clone())?;
118   settings.is_admin(&info.sender)?;
119   let prop = Proposal::new(
```

```
120     &settings,
121     &info.sender,
122     proposal,
123     env.block
124       .time
125       .plus_seconds(conf.proposal_expiration_duration_secs),
126   );
127   let prop_id = next_id(deps.storage)?;
128   PROPOSALS.save(deps.storage, (contract_addr.clone(), prop_id), &
 ↳ prop)?;
129
130   Ok(Response::default().add_event(events::NewProposal::new(
 ↳ contract_addr, prop_id).into()))
131 }
```

Listing 3: contracts/deployer/src/state.rs (Lines 58,64)

```
57 pub fn new(
58   settings: &GovernanceSettings,
59   proposer: impl Into<String>,
60   msg: ProposalMsg,
61   expiration_time: Timestamp,
62 ) -> Self {
63   Self {
64     admins: settings.admins.clone(),
65     required_quorum: settings.quorum,
66     proposer: proposer.into(),
67     msg,
68     expiration_time,
69     accept_votes: vec![],
70     reject_votes: vec![],
71     status: ProposalStatus::InProgress,
72   }
73 }
```

vote function verifies if the owner belongs to the list of owners registered during the proposal creation:

Listing 4: contracts/deployer/src/state.rs (Lines 103-108)

```
86 pub fn vote(
87   &mut self,
```

```
88    current_time: Timestamp,
89    voter: String,
90    is_reject_vote: bool,
91  ) -> Result<(), ContractError> {
92    // we check if the proposal is still in progress
93    if self.status != ProposalStatus::InProgress {
94      return Err(ContractError::ProposalExpired);
95    }
96
97    // proposal in progress but  it has expired
98    if self.expiration_time <= current_time {
99      self.status = ProposalStatus::Expired;
100     return Ok(());
101   }
102   // we need to check if the voter is an admin
103   if !self.admins.contains(&voter) {
104     return Err(ContractError::Unauthorized(format!(
105       "{} not an admin",
106       &voter
107     )));
108   }
```

Risk Level:

Likelihood - 5
Impact - 3

Recommendation:

It is recommended to update the logic of the voting process with one of the following options:

- Rely on indexes of the list of owners (assuming its size won't change) for handling voting, instead of using owners' addresses, which can vary over the time.
- Restrict ownership transfers if a proposal is active. In order to avoid a potential denial-of-service, it's also advisable to limit the number of proposals an owner can submit for a specific interval.

Remediation plan:

SOLVED: The Archway team stated that the fact that a new owner should not be able to vote on old proposals is an expected behavior, but not the opposite scenario. That's why they decided to cover the case in which an adversary with stolen keys might be able to vote despite the admin rights transferal. The issue was fixed in commit e660e83.

3.3 (HAL-03) LIST OF OWNERS COULD HAVE DUPLICATE ENTRIES - MEDIUM

Description:

execute_deploy and execute_transfer_admin functions in **deployer** contract do not restrict if a duplicate entry is present in the list of owners. As a consequence, this situation allows a duplicate owner to have more voting power, as shown in the example below:

- **Initial list of owners:** A, B, C , A
- A calls execute_transfer_admin function to transfer ownership to another address he controls, e.g.: A'
- **Final list of owners:** A', B, C, A
- A has 50% of voting power (A and A') in future proposals

Code Location:

execute_deploy function does not validate if admins vector has duplicate values before saving it to storage:

Listing 5: contracts/deployer/src/contract.rs (Line 176)

```
155 fn execute_deploy(
156  deps: DepsMut,
157  env: Env,
158  info: MessageInfo,
159  code_id: u64,
160  admins: Vec<String>,
161  quorum: Decimal,
162  label: String,
163  msg: Binary,
164 ) -> Result<Response, ContractError> {
165  // validate owner
166  for owner in &admins {
167   deps.api.addr_validate(owner)?;
168  }
169
```

```
170  if quorum > Decimal::percent(100) && quorum < Decimal::percent(0)
 ↳  {
171   return Err(ContractError::BadRequest(
172    "invalid quorum value, must be a percentage".to_string(),
173   ));
174  }
175
176  TRANSIENT_CONTRACT_GOVERNANCE.save(deps.storage, &
 ↳ GovernanceSettings { quorum, admins })?;
```

execute_transfer_admin function does not validate if new_admin address generates a duplicate in the list of owners:

Listing 6: contracts/deployer/src/contract.rs (Line 148)

```
133 fn execute_transfer_admin(
134   deps: DepsMut,
135   _env: Env,
136   info: MessageInfo,
137   contract_addr: String,
138   new_admin: String,
139 ) -> Result<Response, ContractError> {
140   deps.api.addr_validate(&new_admin)?;
141   CONTRACT_GOVERNANCE.update(deps.storage, contract_addr, |r| ->
 ↳ StdResult<_> {
142     let mut r = r.ok_or(StdError::not_found("contract address"))?;
143     let pos = r
144       .admins
145       .iter()
146       .position(|r| *r == info.sender.to_string())
147       .ok_or(StdError::generic_err("unauthorized"))?;
148     r.admins[pos] = new_admin;
149     Ok(r)
150   })?;
```

Risk Level:

Likelihood - 3
Impact - 3

Recommendation:

It is recommended that execute_deploy and execute_transfer_admin functions verify that the list of owners does not have duplicate entries before further execution.

Remediation plan:

SOLVED: The issue was fixed in commit 9333570.

3.4 (HAL-04) OWNERSHIP CAN BE TRANSFERRED WITHOUT CONFIRMATION - LOW

Description:

An incorrect use of the execute_transfer_admin function in **deployer** contract can set owners of vesting contracts to an invalid address and inadvertently lose their corresponding voting power for the migration process, which cannot be undone in any way.

Currently, owners of vesting contracts can transfer **their own address** using the aforementioned function in a single transaction and without confirmation from the new address.

Code Location:

execute_transfer_admin function updates the address of the calling owner in a single step:

Listing 7: contracts/deployer/src/contract.rs (Line 148)

```
133 fn execute_transfer_admin(
134   deps: DepsMut,
135   _env: Env,
136   info: MessageInfo,
137   contract_addr: String,
138   new_admin: String,
139 ) -> Result<Response, ContractError> {
140   deps.api.addr_validate(&new_admin)?;
141   CONTRACT_GOVERNANCE.update(deps.storage, contract_addr, |r| ->
 ↳ StdResult<_> {
142     let mut r = r.ok_or(StdError::not_found("contract address"))?;
143     let pos = r
144       .admins
145       .iter()
146       .position(|r| *r == info.sender.to_string())
147       .ok_or(StdError::generic_err("unauthorized"))?;
```

```
148         r.admins[pos] = new_admin;
149         Ok(r)
150     })?;
151
152     Ok(Response::default())
153 }
```

Risk Level:

Likelihood - 2
Impact - 3

Recommendation:

It is recommended to split **ownership transfer** functionality into execute_transfer_admin and execute_accept_ownership functions. The latter function allows the transfer to be completed by the recipient.

Remediation plan:

RISK ACCEPTED: The Archway team accepted the risk of this finding.

3.5 (HAL-05) LIST OF OWNERS COULD BE EMPTY - LOW

Description:

execute_deploy function in **deployer** contract does not verify if admins vector is empty or not before saving it to storage. As a consequence, if a contract is (mistakenly) deployed with an empty list of owners, no one will be able to handle the migration process of this contract in the future.

Code Location:

Listing 8: contracts/deployer/src/contract.rs (Line 176)

```
155 fn execute_deploy(
156  deps: DepsMut,
157  env: Env,
158  info: MessageInfo,
159  code_id: u64,
160  admins: Vec<String>,
161  quorum: Decimal,
162  label: String,
163  msg: Binary,
164 ) -> Result<Response, ContractError> {
165  // validate owner
166  for owner in &admins {
167   deps.api.addr_validate(owner)?;
168  }
169
170  if quorum > Decimal::percent(100) && quorum < Decimal::percent(0)
 ↳ {
171   return Err(ContractError::BadRequest(
172    "invalid quorum value, must be a percentage".to_string(),
173   ));
174  }
175
176  TRANSIENT_CONTRACT_GOVERNANCE.save(deps.storage, &
 ↳ GovernanceSettings { quorum, admins })?;
```

Risk Level:

Likelihood - 2
Impact - 3

Recommendation:

It is recommended that `execute_deploy` function verifies if the list of owners is not empty before further execution.

Remediation plan:

SOLVED: The issue was fixed in commit 3106f1f.

3.6 (HAL-06) USING COUNTERS INSTEAD OF LISTS COULD SAVE SOME GAS ON PROPOSAL VOTING - INFORMATIONAL

Description:

vote function in **deployer** contract handles lists of reject_votes and accept_votes to check whether a quorum has been reached or not on a proposal. As an optimization technique, if it isn't mandatory to track who voted for or against a proposal, having counters instead of using lists could save some gas on the proposal voting process.

Code Location:

Listing 9: contracts/deployer/src/state.rs (Lines 120-121)

```
118 // we add the vote
119 match is_reject_vote {
120   true => self.reject_votes.push(voter),
121   false => self.accept_votes.push(voter),
122 };
123
124 // we check if quorum was reached
125 if !self.quorum_reached(is_reject_vote) {
126   return Ok(());
127 }
```

Risk Level:

Likelihood - 1
Impact - 1

Recommendation:

It is recommended that vote function uses counters to store the amount of upvotes / votes against, and also to calculate when a quorum has been reached on a proposal.

Remediation plan:

ACKNOWLEDGED: The Archway team acknowledged this finding.

3.7 (HAL-07) UNIMPLEMENTED FUNCTION FOR PROPOSALS QUERYING - INFORMATIONAL

Description:

QueryMsg::Proposals message in **deployer** contract does not implement any internal logic, but invokes the todo! macro. If a user tries to query about the proposals using the message mentioned above, it'll always panic.

Although this situation is not security-related, it is worth noting that could mislead users; hence it is highlighted as an **informational issue**.

Code Location:

Listing 10: contracts/deployer/src/contract.rs (Line 227)

```
217 #[entry_point]
218 fn query(deps: Deps, _env: Env, msg: QueryMsg) -> StdResult<Binary
 ↳ > {
219  match msg {
220   QueryMsg::Settings { contract } => {
221    to_binary(&CONTRACT_GOVERNANCE.load(deps.storage, contract)?)
222   }
223   QueryMsg::Proposal { contract, id } => {
224    to_binary(&PROPOSALS.load(deps.storage, (contract, id))?)
225   }
226   QueryMsg::Proposals { .. } => {
227    todo!()
228   }
229  }
230 }
```

Risk Level:

Likelihood - 1
Impact - 1

Recommendation:

It is recommended to implement the internal logic of `QueryMsg::Proposals` message or do not include the message at all in the final version of the contract.

Remediation plan:

PENDING: The Archway team stated that this issue will be resolved, but in a later iteration.

THANK YOU FOR CHOOSING

// HALBORN