Audit Report

Nym Mixnet and Vesting Contracts

v1.0

March 27, 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 Nym Technologies SA to perform a security audit of Nym Mixnet and Vesting contracts.

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/nymtech/nym/

Commit hash: `d9b682310669658756df974ef3e74c63cf4e7c70`

The scope of this audit is limited to the following directories:

- `contracts/mixnet`
- `contracts/vesting`
- relevant imports by these contracts

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

The audit scope comprehends the Nym Mixnet and the Nym Vesting contracts.

The Nym Mixnet is a multi-purpose mixnet that prevents traffic analysis by an adversary capable of watching the entire network, the audited contract implements its mixnodes staking and reward mechanism.

The Nym Vesting contract allows the contract owner to create vested accounts and enables them to interact with the mixnet like regular accounts.

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.

Code Quality Criteria

The auditor team assesses the codebase's code quality criteria as follows:

Criteria	Status	Comment
Code complexity	Low-Medium	-
Code readability and clarity	High	-
Level of documentation	Medium	No technical documentation was provided for the contracts.
Test coverage	Medium-High	94.42% test coverage for mixnet contract. 66.21% test coverage for vesting contract.

Summary of Findings

No	Description	Severity	Status
1	A failing epoch or interval event execution permanently blocks epoch advancement	Critical	Acknowledged
2	Attackers can force orphan `mixnodes` to join a family without their consent in order to aggregate a large quantity of them in the same `layer`	Critical	Resolved
3	Unbounded iteration in `TrackUndelegation` message handling could make the user unable to undelegate, which also permanently inhibits epoch advancement	Critical	Resolved
4	An attacker can frontrun `BondMixnodeOnBehalf` and `CreateFamilyOnBehalf` messages and modify their payload	Critical	Resolved
5	Signatures can be replayed within "on behalf" transactions in order to impersonate users	Critical	Resolved
6	Key generation for `(owner, proxy)` pairs could lead to collisions	Critical	Resolved
7	`track_delegation` can overwrite existing delegation if more then one is processed within the same block	Critical	Resolved
8	Bond owner is not able to perform operations if the bond is created "on behalf" by a `proxy`	Major	Acknowledged
9	`LeaveFamily` and `LeaveFamilyOnBehalf` messages execution could fail if a significant amount of mixnodes are members of a family	Major	Resolved
10	Mixnet contract is not resilient to transaction reordering performed by validators, allowing validators to extract value	Minor	Resolved
11	Epoch advance mechanism can fail if a significant amount of elements is in the event queue	Minor	Acknowledged
12	Contracts are not compliant with `CW2` Migration specification	Minor	Resolved
13	Nym API is a single point of failure for the Nym	Minor	Acknowledged

	mixnet		
14	`epochs_in_interval` and `epoch_duration` could lead to a “division by zero” error if not correctly validated	Minor	Resolved
15	Contract ownership cannot be transferred	Minor	Acknowledged
16	Number of nodes in each layer is not enforced to be greater than zero	Minor	Resolved
17	Overflow checks not enabled for release profile	Informational	Resolved
18	Custom access controls implementation	Informational	Acknowledged
19	Control flow could reach not implemented functionality	Informational	Acknowledged

Detailed Findings

1. A failing epoch or interval event execution permanently blocks epoch advancement

Severity: Critical

In `contracts/mixnet/src/interval/transactions.rs:56` and `98`, a loop iterates over all pending epoch and interval events during the handling of `AdvanceCurrentEpoch` messages.

Since this iteration is executed in sequence and directly in the transaction context without the use of submessages, a failing event reverts the transaction and permanently inhibits the event execution mechanism.

Consequently, it will not be possible anymore to compute and advance epochs and intervals, resulting in the protocol being stuck.

Recommendation

We recommend wrapping event execution in sub-messages using a reply policy in order to handle failing events gracefully without reverting the whole transaction.

Status: Acknowledged

The client states that they have introduced extra monitoring on the Nym API instances to ensure that epochs are progressing. They have a manual process to progress epochs if needed and they have previously had to do this in mainnet.

2. Attackers can force orphan `mixnodes` to join a family without their consent in order to aggregate a large quantity of them in the same `layer`

Severity: Critical

In `contracts/mixnet/src/contract.rs:114`, the `JoinFamilyOnBehalf` message could be used by an attacker to force a `mixnode` to join a family without its consent.

In fact, the attacker could create a new family and then send a `JoinFamilyOnBehalf` message containing the victim `mixnode identity_key` signed with its private key as the `signature` attribute.

Victim `mixnodes` cannot leave the malicious family without the consent of the attacker (its signature is needed) and cannot join any other family. This implies that an attacker could forcibly add all the orphan `mixnodes` to its malicious family.

Since all mixnodes in a family are organized in the same layer, and the number of mixnodes that have not joined a family could be significant, this would result in an accumulation of nodes in the same layer augmenting the likelihood for attacker to have a mixnode selected for routing packets in the other two layers.

A test case demonstrating how an attacker can add a mixnode to its family without consent can be found in Appendix 1.

Recommendation

We recommend requiring the consent of the mixnode owner to join a family.

Status: Resolved

3. Unbounded iteration in `TrackUndelegation` message handling could make the user unable to undelegate, which also permanently inhibits epoch advancement

Severity: Critical

The `TrackUndelegation` message defined in `contracts/vesting/src/contract.rs:120`, allows the `mixnet` contract to notify the `vesting` contract that a vested account has undelegated part of its stake.

During the message handling, the `remove_delegations_for_mix` function iterates through all the delegation of the specified `mix_id` contained in the `DELEGATIONS` map and then removes them from the storage one by one, resulting in an `O(n)` asymptotic complexity.

This implies that the handler could run out of gas and revert if the account has a significant number of delegations registered in the contract, leading to the impossibility of the user to undelegate.

Also, since the execution flow is initiated from an `Undelegate` event in `PENDING_EPOCH_EVENTS` during the handling of a `AdvanceCurrentEpoch` message, an error raised in the `TrackUndelegation` message as described here will cause epoch advancement to revert, leaving the protocol to be permanently stuck on the current epoch.

Recommendation

We recommend reworking the `remove_delegations_for_mix` function in order to not iterate through all the `mixnode` delegations.

Status: Resolved

4. An attacker could frontrun `BondMixnodeOnBehalf` and `CreateFamilyOnBehalf` messages and modify their payload

Severity: Critical

The `BondMixNodeOnBehalf` and `CreateFamilyOnBehalf` messages both require a valid signature of the given owner address.

However, the signed data does not include the payload of these messages, namely the `cost_params` for `BondMixNodeOnBehalf` and the `label` for `CreateFamilyOnBehalf`.

Therefore, an attacker that sees these transactions in the mempool can extract the signature and resubmit a new one changing its parameters.

In the case of `BondMixNodeOnBehalf`, this is even worse, because the proxy is set to `info.sender`. This means that an attacker is not only able to change the parameters but also set itself as the proxy and have full control over the registered bond.

Recommendation

We recommend including the payload in the signed data.

Frontrunning will still be possible, but it will not have any negative consequences because an attacker is not able to change any parameters. Note that for `BondMixNodeOnBehalf`, `info.sender` is also part of the payload, meaning that only one address should be able to use a given signature.

Status: Resolved

The client implemented a guard allowing only the `vesting` contract to execute “on behalf” messages.

5. Signatures can be replayed within “on behalf” transactions in order to impersonate users

Severity: Critical

The signatures used for verifying the node identity and the family, defined respectively in `contracts/mixnet/src/support/helpers.rs:197` and `212` only include raw data (the owner or the family member) and no additional metadata such as nonces or the message that this signature is intended for.

This enables signature replay attacks across two dimensions:

- Signatures that were intended for one message can be used for other messages. For example, an attacker can take the signature that was used for

`JoinFamilyOnBehalf` and use it with a `LeaveFamilyOnBehalf` message, which forcibly removes a member from the family.
- Signatures for one message type can be used multiple times. For instance, when someone joins the family and is removed later, they can reuse the same signature and join the family again with `JoinFamilyOnBehalf`.

Recommendation

We recommend including nonces in the signed payload or store processed signatures in a map to protect against signature replays. Furthermore, the purpose of the signature (i.e., the message type) should be included in the signed data such that it is not possible to use them for an unintended purpose.

Status: Resolved

The client implemented a guard allowing only the `vesting` contract to execute “on behalf” messages.

6. Key generation for `(owner, proxy)` pairs could lead to collisions

Severity: Critical

In `contracts/mixnet/src/interval/pending_events.rs:94`, the `generate_owner_storage_subkey` function generates a unique key for a given pair of `(owner, proxy)` addresses by calculating the `XOR` of the bytes of their addresses and encoding the result with the `bs58` encoding scheme.

It is possible for two different pairs of `(owner, proxy)` to result in the same `XOR` computation result and therefore result in the same key, leading to a collision and overwritten data.

An example is provided in Appendix 2.

Recommendation:

We recommend storing an incremental counter in the contract and adding it to the key. We also recommend returning an error if data would be overwritten.

Status: Resolved

7. `track_delegation` can overwrite existing delegation if more than one is processed within the same block

Severity: Critical

The function `track_delegation` in `contracts/vesting/src/vesting/account/delegating_account.rs:112` calls the `save_delegation` function with a key that is composed of the `storage` key, the `mix ID` and the block timestamp.

While this key is unique when the delegations happen in different blocks, key collisions can occur if there are multiple delegations in the same block.

In such a scenario, `save_delegation` would overwrite an existing delegation leading to the loss of the previous one.

Recommendation

We recommend checking if the delegation key exists and returning an error in that case to prevent overwriting. Otherwise, a different key scheme that supports multiple delegations in the same block could be considered.

Status: Resolved

8. Bond owner is not able to control their bond if it was created “on behalf” by a `proxy`

Severity: Major

In

- `contracts/mixnet/src/mixnodes/transactions.rs:184`,
- `contracts/mixnet/src/mixnodes/transactions.rs:235`,
- `contracts/mixnet/src/mixnodes/transactions.rs:347`,
- `contracts/mixnet/src/mixnodes/transactions.rs:293`, and
- `contracts/mixnet/src/gateways/transactions.rs:140-147`,

while handling bond related messages, an error is returned if the provided transaction `proxy` attribute is not equal to the stored one.

This behavior has the side effect of also returning an error when there is a stored `proxy`, the bond has been created “on behalf”, but the current transaction has been initiated by the actual bond `owner`.

This implies that the bond `owner` is not able to perform any operation on their bond, which is only controlled by the `proxy`.

Since the `proxy` account could be compromised or lost and there is no functionality that enables the bond `owner` to change it, this could lead to a situation where the `owner` is not able to control their bond.

Recommendation

We recommend allowing the `owner` to perform operations on their bond also if a `proxy` is defined.

Status: Acknowledged

9. `LeaveFamily` and `LeaveFamilyOnBehalf` messages execution could fail if a significant amount of `mixnodes` are members of a family

Severity: Major

The `get_members` function defined in `contracts/mixnet/src/families/storage.rs:32` is called by the `is_family_member` function during the handling of `LeaveFamily` and `LeaveFamilyOnBehalf` messages in order to check if the provided `mixnode` is a member of a specific family.

Since this function performs an iteration over all elements stored in the `MEMBERS` map which implies an `O(n)` asymptotic complexity, the execution may run out of gas if a significant number of `mixnodes` are members of a family.

This implies that `LeaveFamily` and `LeaveFamilyOnBehalf` messages will always fail and it will be not possible anymore for members to leave a family.

Recommendation

We recommend reworking the `is_family_member` function to directly access the `MEMBERS` map to check the membership of a `mixnode` without performing iterations that could lead to an out-of-gas error.

Status: Resolved

10. Mixnet contract is not resilient to transaction reordering performed by validators, allowing validators to extract value

Severity: Minor

The `mixnet` contract relies on the assumption that the mempool's transaction ordering is not alterable. Since validators can influence the ordering in the mempool, this exposes an attack surface that could be leveraged in multiple scenarios by malicious validators.

Suppose the mempool consists of the following three transactions:

1. `ExecuteMsg::ReconcileEpochEvents` (limit 30)

2. `ExecuteMsg::RewardMixnode`
3. `ExecuteMsg::ReconcileEpochEvents` (limit 20)

The Nym API node processes the event queue in separate `ReconcileEpochEvents` transactions, using the `limit` argument to split the queue execution into smaller chunks.

A validator could submit `PledgeMore`, `UnbondMixnode`, and `BondMixNode` transactions to the event queue and reorder the event queue to extract value.

For example, the malicious validator could reorder the `RewardMixnode` transaction to be in between the two `ReconcileEpochEvents` ones, allowing the validator to increase their pledge for a shorter amount of time and still receive a reward.

Another example is related to the ordering of the `AdvanceCurrentEpoch` and `RewardMixnode` transactions: `AdvanceCurrentEpoch` is called before `RewardMixnode` because of the checks in the `REWARDED_SET` to determine which nodes are eligible for rewards. This set is updated in `AdvanceCurrentEpoch`, which could also process an `unbondMixNode` event. This implies that if a node unbonds, it must wait until the end of the epoch and get rewarded for an epoch less. For example, if a node is bonded for two epochs and tries to `unbond` in the third epoch, it will still be rewarded for two epochs even though it remained bonded for three epochs.

Recommendation:

We recommend implementing a mechanism to ensure correct transaction order during their execution.

Status: Resolved

11. Epoch advance mechanism can fail if a significant amount of elements is in the event queue

Severity: Minor

The `try_advance_epoch` function defined in `contracts/mixnet/src/interval/transactions.rs:202` is responsible for handling `AdvanceCurrentEpoch` messages.

Since this function iterates over all pending events in `PENDING_EPOCH_EVENTS` and `PENDING_INTERVAL_EVENTS`, it may run out of gas if the cardinality of the elements is significant.

Consequently, the epoch may not be able to advance and the protocol is stuck.

We classify this as a minor issue since the `ReconcileEpochEvents` message can be used to process events in the queues in multiple batches in order to recover from this situation.

Note that an attacker could anyway continue to spam the network with messages that add further events in the queues in order to make the `ReconcileEpochEvents` transactions less effective.

Recommendation

We recommend optimizing the `AdvanceCurrentEpoch` message handler and declining new events for the queue when the epoch is elapsed and it is still not advanced to the next one.

Status: Acknowledged

12. Contracts are not compliant with `CW2` Migration specification

Severity: Minor

The contracts in scope of this audit do not adhere to the `CW2` Migration specification standard.

This may lead to unexpected problems during contract migration and code version handling.

Recommendation

We recommend following the `CW2` standard in all the contracts. For reference, see https://docs.cosmwasm.com/docs/1.0/smart-contracts/migration.

Status: Resolved

13. Nym API is a single point of failure for the Nym mixnet

Severity: Minor

The protocol relies on an external actor named `rewarding_validator`, also known as Nym API, which is the only one enabled to call some critical privileged transactions of the `mixnet` contract.

The address of this actor can be initialized or updated by the contract `owner` and the authorization is checked by the `ensure_is_authorized` function. The `mixnet` contract trusts this actor and accepts values without performing on-chain verification.

The implementation of Nym API is provided in the `validator-api` crate, which is out of the scope of this audit, so we only analyzed possible scenarios related to the interaction with the contracts in scope.

This actor is the only account allowed to perform:

- **Advancement of epochs**
  The `rewarding_validator` is the only actor allowed to advance epochs by sending `AdvanceCurrentEpoch` messages. Similarly, it is responsible for rewards

distribution by sending `RewardMixnode` messages.

- **Layer assignments**
  The topology and active set of the mixnet are computed by the Nym API and provided to the contract via the `new_rewarded_set` parameter of the `AdvanceCurrentEpoch` message.
  The order of mixnodes is not shuffled using VRF output and the randomization is performed off-chain using `rand::rngs::OsRng` as defined in `validator-api/src/epoch_operations/mod.rs`.

- **Mixnode's performance measurements for rewards calculation**
  The reward calculation is performed on-chain by the `try_reward_mixnode` function defined in `contracts/mixnet/src/rewards/transactions.rs`. However, the computation is strongly dependent on unvalidated performance factors calculated and received from the Nym API.

The Nym API is a single point of failure for the entire Nym mixnet. Its failure, whether by an attack, network outage or manipulation could lead the network to get stuck: the mixnet would preserve its members and topology for an undetermined time and no rewards would be paid out until it the Nym API is restored.

Also, a malicious manipulation performed by someone who took control of the node or a miscalculation of layers and mixnode performance factors would lead to incorrect information directly reflected on-chain without additional validation, impacting users and operators.

Recommendation

We recommend reducing the protocol dependency on the external centralized Nym API, and instead distributing its responsibilities across the network.

For example, epochs could be advanced by validators by implementing a Cosmos SDK module with a `BeginBlocker` handler that automatically calls the `AdvanceCurrentEpoch`.

Also, mixnode's topology could be determined using VRF output based on performance factors. If the VRF output is generated off-chain, the VRF proof could be attached and checked in the contract.

Status: Acknowledged

14. `epochs_in_interval` and `epoch_duration` could lead to a division-by-zero error if not correctly validated

Severity: Minor

In `contracts/mixnet/src/contract.rs:62`, during the execution of the `instantiate` function, `epochs_in_interval` and `epoch_duration` are stored in the contract without being validated to be strictly positive.

This could lead to a division-by-zero error in `common/cosmwasm-smart-contracts/mixnet-contract/src/interval.rs:177`.

Recommendation

We recommend validating `epochs_in_interval` and `epoch_duration` to be strictly positive.

Status: Resolved

15. Contract ownership cannot be transferred

Severity: Minor

Contracts do not implement an ownership transfer mechanism.

Since the defined `owner` has the right to modify critical contract parameters and execute privileged messages, a compromise of its account would have devastating consequences for the protocol.

It is best practice to offer functionality to transfer the ownership to another account.

Recommendation

We recommend implementing a mechanism to transfer the contract ownership to another account following the practices recommended in the issue "Custom access control implementation" below.

Status: Acknowledged

The client states that they plan to manage contract admin settings via governance where the validators vote on proposals to change contract settings.

16. Number of nodes in each layer is not enforced to be greater than zero

Severity: Minor

During the execution of the `try_advance_epoch` function defined in `contracts/mixnet/src/interval/transactions.rs`, mixnodes are assigned to particular positions in the mixnet.

Those positions are communicated by the Nym API actor to the contract using the `LayerAssignment` structure. The contract assumes there are three layers, and according to the docs, a lesser amount of layers would lead to reduced privacy provided to users. However, the final size of a layer is not checked to be greater than zero.

Recommendation

We recommend enforcing that each layer of the mixnet has at least one assigned `mixnode`.

Status: Resolved

17. Overflow checks not enabled for release profile

Severity: Informational

The following contracts do not enable `overflow-checks` for the release profile:

- `contracts/mixnet/Cargo.toml`
- `contracts/vesting/Cargo.toml`

While enabled implicitly through the workspace manifest, a future refactoring might break this assumption.

Recommendation

We recommend enabling overflow checks in all packages, including those that do not currently perform calculations, to prevent unintended consequences if changes are added in future releases or during refactoring. Note that enabling overflow checks in packages other than the workspace manifest will lead to compiler warnings.

Status: Resolved

18. Custom access controls implementation

Severity: Informational

Contracts implement custom access controls. Although no instances of broken controls or bypasses have been found, using a battle-tested implementation reduces potential risks and the complexity of the codebase.

Also, the access control logic is duplicated across the handlers of each function, which negatively impacts the code's readability and maintainability.

Recommendation

We recommend using a well-known access control implementation such as `cw_controllers::Admin` (https://docs.rs/cw-controllers/latest/cw_controllers/struct.Admin.html).

Status: Acknowledged

19. Control flow could reach not implemented functionality

Severity: Informational

The `KickFamilyMember` message, handled by the `_try_head_kick_member` function defined at `contracts/mixnet/src/families/transactions.rs:230`, returns a `MixnetContractError::NotImplemented` error.

Attempts to use this functionality will always fail.

Recommendation

We recommend removing entrypoints to functionality that is not implemented.

Status: Acknowledged

Appendix: Test Cases

1. Test case for “An attacker could force orphan mixnodes to join its family without their consent in order to aggregate a large quantity of them in the same layer”

```
#[test]
    fn test_family_without_consent() {
        let mut deps = test_helpers::init_contract();
        let env = mock_env();
        let mut rng = test_helpers::test_rng();

        let head = "alice";
        let malicious_head = "timmy";

        let minimum_pledge =
minimum_mixnode_pledge(deps.as_ref().storage).unwrap();

        let (head_mixnode, head_sig, _head_keypair) =
            test_helpers::mixnode_with_signature(&mut rng, head);

        let (malicious_mixnode, malicious_sig, malicious_keypair) =
            test_helpers::mixnode_with_signature(&mut rng,
malicious_head);

        let cost_params = fixtures::mix_node_cost_params_fixture();

        let member = "bob";
        let (member_mixnode, member_sig, _) =
            test_helpers::mixnode_with_signature(&mut rng, member);

        // we are informed that we didn't send enough funds
        crate::mixnodes::transactions::try_add_mixnode(
            deps.as_mut(),
            env.clone(),
            mock_info(head, &[minimum_pledge.clone()]),
            head_mixnode.clone(),
            cost_params.clone(),
            head_sig.clone(),
        )
        .unwrap();
        crate::mixnodes::transactions::try_add_mixnode(
            deps.as_mut(),
            env.clone(),
            mock_info(malicious_head, &[minimum_pledge.clone()]),
            malicious_mixnode.clone(),
            cost_params.clone(),
            malicious_sig.clone(),
        )
        .unwrap();

        crate::mixnodes::transactions::try_add_mixnode(
            deps.as_mut(),
            env.clone(),
            mock_info(member, &[minimum_pledge.clone()]),
            member_mixnode.clone(),
            cost_params.clone(),
            member_sig.clone(),
        )
        .unwrap();

        try_create_family(
            deps.as_mut(),
            mock_info(malicious_head.clone(), &[]),
            malicious_sig.clone(),
            "malicious",
        ).unwrap();
        let family_head =
FamilyHead::new(&malicious_mixnode.identity_key);
        assert!(get_family(&family_head, &deps.storage).is_ok());

        let family = get_family_by_label("malicious",
&deps.storage).unwrap();
        assert!(family.is_some());
        assert_eq!(family.unwrap().head_identity(),
family_head.identity());

        let family = get_family_by_head(family_head.identity(),
&deps.storage).unwrap();
        assert_eq!(family.head_identity(), family_head.identity());

        let join_signature = malicious_keypair
            .private_key()
            .sign(member_mixnode.identity_key.as_bytes())
            .to_base58_string();
        try_join_family_on_behalf(
            deps.as_mut(),
            mock_info(member, &[]),
            member.to_string(),
            join_signature.clone(),
            malicious_mixnode.identity_key.clone()

        )
        .unwrap();

        let family = get_family(&family_head, &deps.storage).unwrap();

        assert!(is_family_member(&deps.storage, &family,
&member_mixnode.identity_key).unwrap());

        try_leave_family(
            deps.as_mut(),
            mock_info(member, &[]),
            join_signature.clone(),
            malicious_mixnode.identity_key.clone(),
        )
        .unwrap();

        let family = get_family(&family_head, &deps.storage).unwrap();
        assert!(!is_family_member(&deps.storage, &family,
&member_mixnode.identity_key).unwrap());
    }
```

2. Real data example for “Possibility of key duplication for different (owner, proxy) pairs”

```
Owner address: 0b00001111001100010100
Proxy address: 0b00000101110001010010
XOR value:     0b00001010111101000110

Owner address: 0b00001111001100010101
Proxy address: 0b00000101110001010011
XOR value:     0b00001010111101000110
```