



# **Security Audit Report**

Archway: Q3 2023

Authors: Darko Deuric, Marko Juric

Last revised 12 October, 2023

# Table of Contents

<b>Audit Overview .....</b>	<b>1</b>
The Project	1
Scope of this audit	1
Conducted work	1
Conclusions	1
<b>Audit Dashboard .....</b>	<b>2</b>
Target Summary	2
Engagement Summary	2
Severity Summary	2
<b>System Overview.....</b>	<b>3</b>
Withdrawal Contract	3
Vesting Contract	3
The why of Withdrawal Contract	3
Fairness Model	3
<b>Threat inspection .....</b>	<b>5</b>
Threat Model Entry: The funder is unable to clawback	5
Threat Model Entry: Multiple undelegations and storage handling cause inconsistencies on reply mechanism	6
Threat Model Entry: MaxEntries no. of undelegations prevent funder from doing clawback	8
Threat Model Entry: Unintented outcome while clawback, unbonding and slashing are happening consecutively	8
<b>Findings .....</b>	<b>9</b>
Lack of documentation	10
Redundant Query in Loop	11
Possible blocking of clawback by Beneficiary	13
Possible race for portion between Funder and Beneficiary	15
<b>Appendix: Vulnerability Classification .....</b>	<b>18</b>
Impact Score	18
Exploitability Score	18
Severity Score	19
<b>Disclaimer .....</b>	<b>21</b>

# Audit Overview

## The Project

In September 2023, the Archway development team engaged Informal Systems to conduct a security audit of the Vesting system. This system facilitates the vesting of tokens for designated individuals (beneficiaries) while incorporating a clawback feature. During the audit, the project was in active use internally by Archway employees and individuals who had vested Archway assets. The overarching goal was to ensure its robustness and security in preparation for a public release, making it accessible to a broader community of developers.

## Scope of this audit

The audit was conducted by the following individuals:

- Darko Deuric
- Marko Juric

The agreed-upon work plan outlined the following key tasks, primarily focusing on evaluating and analyzing the code and specifications related to Vesting contracts:

1. **Vesting Smart Contract:** This contract allows funder to vest tokens to beneficiary with possibility of clawback. Additionally, the contract allows the beneficiary to perform actions such as delegation, undelegation, vote and withdrawal.
2. **Withdrawal Smart Contract:** This contract is responsible for escrowing staking rewards, ensuring that staking rewards are consistently directed to the withdrawal contract. It also aids the vesting contract in accounting for rewards.

## Conducted work

At the kickoff meeting, the Archway team gave us a brief introduction to the Vesting system with the immediate walkthrough of code which needs to be audited. Our team performed high-quality line-by-line manual code review with a focus mainly on code correctness and the critical points analysis specific to each component. Over the shared Slack channel, we discussed all the necessary information for sync meeting arrangements, additional questions, the Github audit repository, etc.

## Conclusions

We were thoroughly impressed by the exceptional quality of the codebase, showcasing a well-organized structure and high readability. The inclusion of both unit and integration tests within the test suite significantly bolstered our confidence in the code's reliability.

As informed by the Archway team in advance, we acknowledge the absence of certain integration tests, particularly those encompassing edge cases involving combinations of scenarios such as delegation, undelegation, slashing, clawback, etc. Our team highly appreciates the insightful discussions regarding the implementation of the vesting system. These discussions have been immensely valuable in providing a deep understanding of the challenges that the Archway team encountered while striving to ensure a fairness model between the funder and beneficiary, particularly caused by the slashing component. We eagerly anticipated proposing a solution to address this critical issue.

Given the codebase's remarkable quality and design, our identified only one issue classified as high severity, with all other observations falling under the category of low and informational.

# Audit Dashboard

## Target Summary

- **Type:** Specification and Implementation
- **Platform:** Rust, Cosmos SDK
- **Artifacts:**
  - Vesting contracts [repo](#), commit: [vesting v2](#)

## Engagement Summary

- **Dates:** 18.09.2023 to 29.09.2023
- **Method:** Manual code review, protocol analysis
- **Employees Engaged:** 2

## Severity Summary

Finding Severity	#
Critical	0
High	1
Medium	0
Low	1
Informational	2
<b>Total</b>	<b>4</b>

## System Overview

The vesting system consists of two contracts: the Vesting contract and the Withdrawal contract. These contracts are designed to handle the vesting of assets, rewards, staking, governance votes and potential clawbacks.

### Withdrawal Contract

- The Withdrawal contract escrows (holds) staking rewards until the `execute_withdraw_staking_rewards` function is called.
- It is instantiated by saving the sender's address as the owner.
- Only the owner (admin) of the Withdrawal contract can call the `execute_withdraw` function, which transfers all the contract balances to a specific target address.
- The owner/admin of the Withdrawal contract is set to be the Vesting contract, ensuring that only the Vesting contract can instruct the Withdrawal contract to forward funds to a specific address.

### Vesting Contract

- The Vesting contract is used by Archway employees and individuals with vested Archway assets.
- It may have a clawbacker (e.g., phi labs) who has the ability to claw back the contract, but this is only possible if the vested assets have matured.
- The contract operates with a cliff period, which is specific to vesting. For example, if the vesting duration is 4 years, the first tranche of assets unlocks after the cliff duration (e.g., 1 year), and 25% of vested coins unlock. The remainder of the coins unlocks linearly over time.
- The Vesting contract allows staking and accepts one coin upon instantiation.
- It creates a vesting state that includes the funder (clawbacker), beneficiary (owner of vested assets), the initial amount, vesting time, cliff time, the percentage of cliffs that unlock, whether staking rewards are liquid, and the unlock time for rewards.
- The Vesting contract also creates a Withdrawal contract and allows setting the withdrawal address.
- It interacts with the staking module, governance, and distribution to manage vesting, rewards, and clawback operations.

### The why of Withdrawal Contract

- The Withdrawal contract is used for accounting reasons and to address issues with the Staking module's behavior, which is not entirely lazy.
- The Staking module automatically sends staking rewards, but the contract is not aware of these transactions, leading to complications in tracking internal balances.
- To address these limitations, all funds are directed to the Withdrawal contract, simplifying the management of staking rewards.
- Edge cases related to Staking and Slashing in bonding delegations that locked the contract funds were identified earlier on by the Phi lab developers.
- If a contract is slashed, the contract is not aware of it, and this separation of staking rewards from the initial amount (vested coins) ensures that staking rewards remain free even after slashing.
- The slashing problem arises when a clawback occurs, and the initial amount is lower due to slashing, affecting calculations for distribution.
- The separation of staking rewards from the initial amount is essential to ensure that clawback and distribution calculations work correctly based on the initial amount.

### Fairness Model

The fairness model for the Vesting and Withdrawal smart contracts is designed to ensure equitable treatment of both the funder (clawbacker) and the beneficiary (owner of vested assets) while accounting for various scenarios, including slashing and clawbacks.

However, there are exceptions and challenges in cases where the contract cannot detect certain events, potentially leading to disparities in outcomes.

## **Slashing and Clawback Contention**

- If slashing occurs and there is a clawback in progress, both the funder and beneficiary enter contention mode.
- In this mode, they compete for the portion of the assets that have been slashed.

## **Contract's Lack of Slashing Detection**

- One of the challenges in maintaining fairness is that the contract cannot directly detect if slashing has occurred.
- Slashing events may happen outside the contract's awareness, impacting the overall asset balance.

## **Delayed Execution**

- Contracts can experience delayed execution, which may happen long after undelegation occurred and after the contract received the undelegation amount.
- Delayed execution events trigger the contract to release spendable funds through an "unspend" operation.

## **Impact on Funder in Case of Slashing:**

- Fairness considerations acknowledge that in the event of slashing, the funder (clawbacker) may face unfavorable outcomes.
- The slashing event reduces the initial amount held by the contract, impacting distribution calculations and potentially diminishing the funder's share although the beneficiary bears responsibility for any actions or conditions that lead to slashing.

# Threat inspection

## Threat Model Entry: The funder is unable to clawback

### Description:

The vesting smart contract allows the `beneficiary` to immediately delegate or spend funds from the initial balance using the `ExecuteMsg::Delegate` function, even before the `cliff_time` is reached. However, a vulnerability arises if the `funder` decides to clawback funds, possibly even before the cliff time, with the intention of reclaiming all the funds.

The condition for the `funder` to withdraw the clawed back funds is that the `beneficiary` must perform an undelegation. However, there is no mechanism in place to ensure that the `beneficiary` performs the undelegation.

### Attack Vector:

1. The `beneficiary` immediately delegates or spends some or all of the funds from the initial balance using the `ExecuteMsg::Delegate` function.
2. The `funder` decides to clawback the funds, even before the cliff time has been reached.
3. The `funder` is unable to withdraw the clawed back funds because the `beneficiary` has not performed the required undelegation.
4. The `beneficiary` does not perform the undelegation and retains control over the delegated funds.
5. In this scenario, neither the `beneficiary` nor the `funder` can withdraw the vested coins, leading to a situation where the `funder` loses access to the funds they intended to reclaim.

**Additional Note:** It's important to note that in this scenario, the `beneficiary` can also receive staking rewards, and no mechanism can prevent this. These rewards are separate from the clawed back funds and are accessible to the `beneficiary` without requiring permission from the `funder`.

Code snippet from unit test:

```
fn withdraw() {
    let mut v = new();
    v.spend(Uint128::new(1_000_000)).unwrap(); // beneficiary spends all
    assert_eq!(Uint128::zero(), v.clawback(ts(100), None)); // clawback before
    cliff time -> funder should get all the money at the end
    assert_eq!(Uint128::zero(), v.funder_withdraw(None)); // funder is unable to
    withdraw
    assert_eq!(Uint128::zero(), v.beneficiary_withdraw(ts(850), None)); //
    beneficiary is unable to withdraw too, but will get staking rewards
}
```

## Threat Inspection Results:

Based on the `code snippet`, it's clear that the threat is not real. The funder can indeed force undelegations in the `execute_clawback` function, eliminating the dependency on the goodwill of the beneficiary to trigger undelegation.

Therefore, the threat of the funder being unable to clawback funds is not valid in the context of the smart contract's implementation.

## Threat Model Entry: Multiple undelegations and storage handling cause inconsistencies on reply mechanism

### Description:

The `execute_undelagate` function in the vesting smart contract exhibits problematic behavior that results in the overwriting of previous undelegation records with each new undelegation request. This likely unintended behavior stems from the following code snippet:

```
UNDELEGATIONS_TMP.save(deps.storage, 0, &amount)?;
```

In this line, a fixed key of `0` is consistently used for all undelegation records. As a result, when multiple consecutive undelegations are triggered by the same `beneficiary`, each new undelegation operation overwrites the previous one with the same key, leading to unintended consequences.

### Attack Vector:

Consider the following scenario:

1. A user triggers `execute_undelagate` three times consecutively with different inputs:
  - `undelegation_1`: Validator `val1`, Amount `amount1`
  - `undelegation_2`: Validator `val2`, Amount `amount2`
  - `undelegation_3`: Validator `val3`, Amount `amount3`
2. Three replies are handled by the `reply_undelagate` function, and each reply attempts to `load` the undelegation `amount` from the `UNDELEGATIONS_TMP` map using the `reply.id`. However, because the code consistently uses `0` as the key in `execute_undelagate`, issues arise:
  - The reply for `undelegation_1` may incorrectly load the `amount` from `undelegation_3` due to the overwriting of records.
  - Subsequent replies (for `undelegation_2` and `undelegation_3`) may fail with panics because the record with `id: 0` was removed, causing issues in loading data.

### Consequences:

1. **Incorrect Mapping:** The `completion_time` and `amount` mapping in the `UNDELEGATIONS` map can become incorrect, leading to inaccurate tracking of undelegation amounts at different completion times.
2. **Panic:** Subsequent replies handling undelegations may fail with panics due to the removal of the record with `id: 0`, causing disruption in the contract execution flow.



Moreover, it's worth noting that this undesirable behavior is avoided inside the `execute_clawback` function. `UNDELEGATIONS_TMP` correctly saves undelegations with updated `ids`, ensuring that each undelegation is associated with a unique identifier:

```
for delegation in deps
  .querier
  .query_all_delegations(env.contract.address)?
  .into_iter()
{
  UNDELEGATIONS_TMP.save(deps.storage, id, &delegation.amount.amount)?;
  undelegations.push(SubMsg::reply_on_success(
    StakingMsg::Undelegate {
      validator: delegation.validator,
      amount: delegation.amount,
    },
    id,
  ));
  id += 1;
}
```

## Threat Inspection Results:

Based on a deeper analysis and understanding of the actor model in cosmwasm and how contracts are executed within it, the threat is false positive.

- Cosmwasm enforces atomicity in contract execution, ensuring that contract operations are isolated and executed atomically without interference from external actors.
- In the actor model of cosmwasm, it is not possible for `undelegation_1` to be separated from the appropriate `reply_undelegate` handling, and there is no risk of reentrancy problems or similar attack vectors.
- As a result, the threat described above, where consecutive undelegations might overwrite records and cause unintended consequences, is not valid within the cosmwasm actor model.

Threat Model Entry: MaxEntries no. of undelegations prevent funder from doing clawback

Description: <https://informalsystems.atlassian.net/wiki/spaces/AQ2/pages/123142147/Possible+blocking+of+clawback+by+malicious+Beneficiary#Description>

Attack Vector: <https://informalsystems.atlassian.net/wiki/spaces/AQ2/pages/123142147/Possible+blocking+of+clawback+by+malicious+Beneficiary#Problem-Scenarios>

Threat Model Entry: Unintented outcome while clawback, unbonding and slashing are happening consecutively

Description: <https://informalsystems.atlassian.net/wiki/spaces/AQ2/pages/133595182/Possible+race+for+portion+between+Funder+and+Beneficiary#Description>

Attack Vector: <https://informalsystems.atlassian.net/wiki/spaces/AQ2/pages/133595182/Possible+race+for+portion+between+Funder+and+Beneficiary#Problem-Scenarios>

## Findings

Title	Type	Severity
Redundant Query in Loop	IMPLEMENTATION	0 INFORMATIONAL
Lack of documentation	DOCUMENTATION	0 INFORMATIONAL
Possible race for portion between Funder and Beneficiary	IMPLEMENTATION	1 LOW
Possible blocking of clawback by Beneficiary	IMPLEMENTATION	3 HIGH

## Lack of documentation

<b>Title</b>	Lack of documentation
<b>Project</b>	Archway: Q3 2023
<b>Type</b>	DOCUMENTATION
<b>Severity</b>	0 INFORMATIONAL
<b>Impact</b>	0 NONE
<b>Exploitability</b>	0 NONE
<b>Issue</b>	

## Description

Before the audit, it was evident that the audit scope lacked sufficient documentation regarding vesting contracts. Additionally, a code walkthrough was deemed necessary for the audit team to gain necessary understanding of the project's structure and processes.

## Recommendation

Our team recommends creating at least a high-level documentation that offers a clear understanding of the project's structure, processes, and functionalities, reducing the onboarding time needed to gain desirable understanding of the project.

## Redundant Query in Loop

<b>Title</b>	Redundant Query in Loop
<b>Project</b>	Archway: Q3 2023
<b>Type</b>	IMPLEMENTATION
<b>Severity</b>	0 INFORMATIONAL
<b>Impact</b>	0 NONE
<b>Exploitability</b>	0 NONE
<b>Issue</b>	

### Involved artifacts

- [contracts/vesting/src/contract.rs](#)

### Description

In the provided [code snippet](#), there is a potential performance issue related to the repeated execution of the `query_all_delegations()` function inside a for loop. This function is called in each iteration of the loop, even though the `delegations` data is not modified within the loop.

### Problem Scenarios

The code snippet in question contains a for loop that iterates over delegations and, for each delegation, calls the `query_all_delegations()` function. The function queries the delegations for the same `env.contract.address` in each iteration.

This could result in unnecessary and redundant calls to the blockchain or external data source to retrieve the same data repeatedly, leading to increased execution time and potential performance degradation.

### Recommendation

To optimize the code and improve performance, you should modify the code to query the delegations outside of the loop and store the result in a variable. Then, iterate over the stored delegations in the loop. This change will reduce the number of redundant queries and improve code efficiency.

```
// Query all delegations once and store the result
let allDelegations = deps.querier.query_all_delegations(env.contract.address)?;

// Iterate over the stored delegations
for delegation in allDelegations.into_iter() {
    // ...
}
```

}

## Possible blocking of clawback by Beneficiary

<b>Title</b>	Possible blocking of clawback by Beneficiary
<b>Project</b>	Archway: Q3 2023
<b>Type</b>	IMPLEMENTATION
<b>Severity</b>	3 HIGH
<b>Impact</b>	2 MEDIUM
<b>Exploitability</b>	3 HIGH
<b>Issue</b>	

### Involved artifacts

- [contracts/vesting/src/contract.rs](#)

### Description

**Function:** `execute_clawback`

In the vesting smart contract, alongside others, the `execute_clawback` function performs the following checks and actions:

1. **Check if Ongoing Delegations Exist:**
  - Verify if there are ongoing delegations executed by the beneficiary on behalf of the vesting contract.
2. **Undelegate Process:**
  - Call the staking module to immediately attempt to undelegate if there are existing delegations.

**Cases in Which** `MsgUndelegate` will fail:

- The delegation does not exist.
- The validator associated with the delegation does not exist.
- The delegation has fewer shares than the amount specified ( `Amount` ).
- An existing `UnbondingDelegation` has the maximum allowed entries as defined by `params.MaxEntries`.
- The `Amount` is in a denomination different from the one defined by `params.BondDenom`.

### Problem Scenarios

**Potential problem: manipulation of** `max_entries` parameter

There is a potential issue concerning the manipulation of the `max_entries` parameter defined in the staking module, which represents the maximum entries for either unbonding delegation or redelegation (per pair/trio).

The standard value for `max_entries` on the Cosmos chain is set to seven, implying that there can be up to seven unbonding or redelegation events per 21 days. However, this number might be considered relatively low, and it's conceivable that a beneficiary could inadvertently reach this limit.

On the flip side, a malicious beneficiary could intentionally attempt to maintain the `max_entries` value at the configured maximum by constantly initiating multiple undelegations.

For example, attempting to initiate a new undelegation event after reaching the maximum value of the `max_entries` parameter would result in a `failure`. This scenario could prevent a funder from executing a clawback. As long as the beneficiary is able to keep the `max_entries` value at the exact needed value, the clawback will be blocked, allowing the beneficiary to continue denying the clawback and still earn staking rewards.

## Recommendation

### Possible solutions to prevent the issue:

In discussions with the Archway team, various potential solutions have been identified to mitigate the issue described. These include:

1. **Omitting the `amount` parameter:**
  - Allow the beneficiary to delegate/undelegate the entire available amount to a specific validator, eliminating the possibility of specifying a partial amount.
2. **Implementing a counter:**
  - Introduce a counter to track the usage of the `max_entries` parameter and prevent new undelegations when the limit is reached, ensuring adherence to the maximum entry constraint.
3. **Creating a separate method:**
  - Develop a distinct method that restricts a beneficiary from initiating additional delegations/undelegations, even when the limit defined by `max_entries` is reached. This would effectively block further manipulation.

These are some of the potential approaches, and there may be additional options. It is recommended to thoroughly analyze and choose the solution that aligns best with the specific requirements and circumstances.



## Possible race for portion between Funder and Beneficiary

<b>Title</b>	Possible race for portion between Funder and Beneficiary
<b>Project</b>	Archway: Q3 2023
<b>Type</b>	IMPLEMENTATION
<b>Severity</b>	1 LOW
<b>Impact</b>	2 MEDIUM
<b>Exploitability</b>	1 LOW
<b>Issue</b>	

### Involved artifacts

- [contracts/vesting/src/contract.rs](#)

### Description

During our initial kickoff meeting with the Archway team, we delved into the ongoing accountability challenges tied to the vesting contract. One prominent concern revolves around the potential race condition that may arise between the funder and the beneficiary, each trying to receive their full share of tokens.

The vesting contract serves a critical purpose by allowing the beneficiary to stake funds and maintaining a robust system of accountability for actions such as delegation, undelegation, and withdrawal. Additionally, it provides the funder with the ability to claw back funds if necessary. However, a fundamental issue lies in the contract's inability to detect if it has been slashed, and by what magnitude.

This vulnerability poses significant complications when both the funder and beneficiary intend to withdraw their respective portions. The actual balance within the vesting contract could be lower than expected due to a slash event. Consequently, the individual who initiates the withdrawal first receives the anticipated amount, while the other party incurs a loss due to the slash, obtaining a reduced amount as the contract does not hold the desired balance.

### Problem Scenarios

Lets consider the following scenario which leads to the situation in description:

init\_bal = 1M (after init)  
 cliff\_time = 250s  
 vesting\_period = 1000s  
 slash amount = 50

action	real contract_bal	ben_spent	fund_withdrawn	ben_withdrawn	fund_clawback
initialization	1M	0	0	0	0
delegate(600)	400	600	0	0	0
300s passed					
beneficiary withdraw	100	600	0	300	0
clawback(600s)	0	600	100	300	400
slashing(50)					
Unb time passed	550	0	100	300	400
800s passed					
beneficiary withdraw	250	0	100	600	400
funder withdraw	0	0	350	600	400

- Initialization:**
  - Vesting contract instantiated with an initial amount of 1M tokens.
- Beneficiary delegation:**
  - Beneficiary delegates 600 tokens.
- Beneficiary withdraws:**
  - Beneficiary withdraws 300 tokens which were unlocked after 300s.
- Funder initiates clawback:**
  - Funder clawbacks with immediately initiating unbonding for delegated tokens. At this point funder was able to withdraw 100 tokens
- Slashing occurs:**
  - Vesting contract's validator has been slashed by 50 tokens.
- Unbonding time passes:**
  - Contract receives 550 instead of expected 600 tokens, completely unaware of slashing.
- Beneficiary's withdrawal:**
  - After total 800s passed, beneficiary withdraws all unlocked funds, which is 300.
- Funder's withdrawal:**
  - Funder withdraws 250 instead of expected 300 tokens, because of slashing event earlier.

In conclusion, the order in which the funder and the beneficiary withdraw their tokens impacts the fairness of the distribution. If slashing occurs during unbonding, the contract loses track of accountability, leading to an unfair distribution of tokens to either the funder or the beneficiary. Additionally, the extent of unfairness is increasing with a higher slashing portion.

## Recommendation

As above mentioned, in our kickoff meeting with the Archway team, they informed us about critical issue regarding accountability in the vesting contract implementation. The main concern revolves around the potential unfairness in the token distribution, especially when slashing occurs. The Archway team recognized this as a primary challenge inherent in the current implementation.

Upon further analysis and discussions with the Archway team, it was established that the beneficiary should bear the responsibility for slashing. This is because the beneficiary is the one who initially delegated funds to the

validator that faced slashing. As a result, the funder should not be adversely affected and should not bear the consequences of potential slashing.

The Archway team expressed reluctance to resolve this issue by adding further complexity to the existing contract, which is already at a reasonable level of complexity. Additionally, augmenting the codebase might lead to increased gas costs, which is an undesirable outcome.

Given these circumstances, our team proposes a practical solution. We suggest preventing the beneficiary from withdrawing their portion until the funder has completed their withdrawal. For instance, if the funder initiates a clawback, triggering undelegations, the contract could enforce a sequence where the funder must withdraw first. To ensure fairness, a defined time period could be set during which the funder has the exclusive right to withdraw. If the funder doesn't utilize this right within the specified timeframe, the beneficiary can proceed with their withdrawal.

This approach aims to strike a balance of fairness, ensuring that the funder is not negatively impacted due to slashing resulting from the beneficiary's choice of a validator. By implementing this solution, we aim to address the issue and maintain an equitable distribution of tokens in the vesting contract.


## Appendix: Vulnerability Classification

For classifying vulnerabilities identified in the findings of this report, we employ the simplified version of [Common Vulnerability Scoring System \(CVSS\) v3.1](#), which is an industry standard vulnerability metric. For each identified vulnerability we assess the scores from the *Base Metric Group*, the [Impact score](#), and the [Exploitability score](#). The *Exploitability score* reflects the ease and technical means by which the vulnerability can be exploited. That is, it represents characteristics of the *thing that is vulnerable*, which we refer to formally as the *vulnerable component*. The *Impact score* reflects the direct consequence of a successful exploit, and represents the consequence to the *thing that suffers the impact*, which we refer to formally as the *impacted component*. In order to ease score understanding, we employ [CVSS Qualitative Severity Rating Scale](#), and abstract numerical scores into the textual representation; we construct the final *Severity score* based on the combination of the Impact and Exploitability sub-scores.

As blockchains are a fast evolving field, we evaluate the scores not only for the present state of the system, but also for the state that deems achievable within 1 year of projected system evolution. E.g., if at present the system interacts with 1-2 other blockchains, but plans to expand interaction to 10-20 within the next year, we evaluate the impact, exploitability, and severity scores wrt. the latter state, in order to give the system designers better understanding of the vulnerabilities that need to be addressed in the near future.

### Impact Score

The Impact score captures the effects of a successfully exploited vulnerability on the component that suffers the worst outcome that is most directly and predictably associated with the attack.

Impact Score	Examples
<b>High</b>	Halting of the chain; loss, locking, or unauthorized withdrawal of funds of many users; arbitrary transaction execution; forging of user messages / circumvention of authorization logic
<b>Medium</b>	Temporary denial of service / substantial unexpected delays in processing user requests (e.g. many hours/days); loss, locking, or unauthorized withdrawal of funds of a single user / few users; failures during transaction execution (e.g. out of gas errors); substantial increase in node computational requirements (e.g. 10x)
<b>Low</b>	Transient unexpected delays in processing user requests (e.g. minutes/a few hours); Medium increase in node computational requirements (e.g. 2x); any kind of problem that affects end users, but can be repaired by manual intervention (e.g. a special transaction)
 <b>None</b>	Small increase in node computational requirements (e.g. 20%); code inefficiencies; bad code practices; lack/incompleteness of tests; lack/incompleteness of documentation

### Exploitability Score

The Exploitability score reflects the ease and technical means by which the vulnerability can be exploited; it represents the characteristics of the vulnerable component. In the below table we list, for each category, examples of actions by actors that are enough to trigger the exploit. In the examples below:

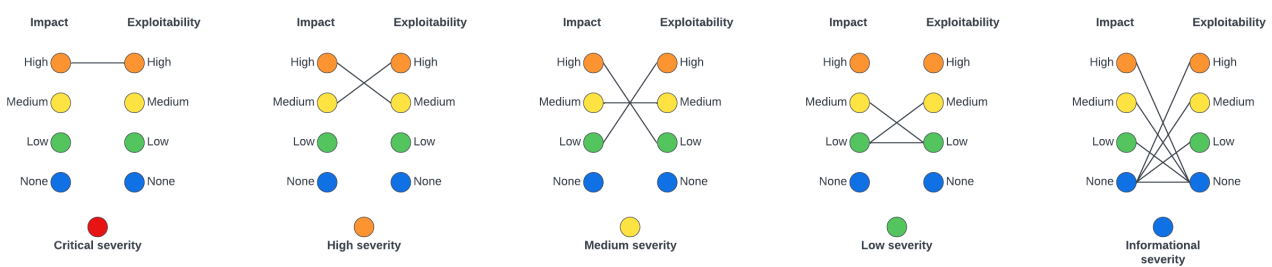
- *Actors* can be any entity that interacts with the system: other blockchains, system users, validators, relayers, but also uncontrollable phenomena (e.g. network delays or partitions).
- *Actions* can be

- *legitimate*, e.g. submission of a transaction that follows protocol rules by a user; delegation/redelegation/bonding/unbonding; validator downtime; validator voting on a single, but alternative block; delays in relaying certain messages, or speeding up relaying other messages;
- *illegitimate*, e.g. submission of a specially crafted transaction (not following the protocol, or e.g. with large/incorrect values); voting on two different alternative blocks; alteration of relayed messages.
- We employ also a *qualitative measure* representing the amount of certain class of power (e.g. possessed tokens, validator power, relayed messages): *small* for < 3%; *medium* for 3-10%; *large* for 10-33%, *all* for >33%. We further quantify this qualitative measure as relative to the largest of the system components. (e.g. when two blockchains are interacting, one with a large capitalization, and another with a small capitalization, we employ *small* wrt. the number of tokens held, if it is small wrt. the large blockchain, even if it is large wrt. the small blockchain)

Exploitability Score	Examples
<b>High</b>	illegitimate actions taken by a small group of actors; possibly coordinated with legitimate actions taken by a medium group of actors
<b>Medium</b>	illegitimate actions taken by a medium group of actors; possibly coordinated with legitimate actions taken by a large group of actors
<b>Low</b>	illegitimate actions taken by a large group of actors; possibly coordinated with legitimate actions taken by all actors
● <b>None</b>	illegitimate actions taken in a coordinated fashion by all actors


## Severity Score

The severity score combines the above two sub-scores into a single value, and roughly represents the probability of the system suffering a severe impact with time; thus it also represents the measure of the urgency or order in which vulnerabilities need to be addressed. We assess the severity according to the combination scheme represented graphically below.



As can be seen from the image above, only a combination of high impact with high exploitability results in a Critical severity score; such vulnerabilities need to be addressed ASAP. Accordingly, High severity score receive vulnerabilities with the combination of high impact and medium exploitability, or medium impact, but high exploitability.

Severity Score	Examples
● <b>Critical</b>	Halting of chain via a submission of a specially crafted transaction

Severity Score	Examples
<b>High</b>	Permanent loss of user funds via a combination of submitting a specially crafted transaction with delaying of certain messages by a large portion of relayers
<b>Medium</b>	Substantial unexpected delays in processing user requests via a combination of delaying of certain messages by a large group of relayers with coordinated withdrawal of funds by a large group of users
<b>Low</b>	2x increase in node computational requirements via coordinated withdrawal of all user tokens
 <b>Informational</b>	Code inefficiencies; bad code practices; lack/incompleteness of tests; lack/incompleteness of documentation; any exploit for which a coordinated illegitimate action of all actors is necessary

## Disclaimer

This report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability, etc.) set forth in the associated Services Agreement. This report provided in connection with the Services set forth in the Services Agreement shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Agreement.

This audit report is provided on an “as is” basis, with no guarantee of the completeness, accuracy, timeliness or of the results obtained by use of the information provided. Informal has relied upon information and data provided by the client, and is not responsible for any errors or omissions in such information and data or results obtained from the use of that information or conclusions in this report. Informal makes no warranty of any kind, express or implied, regarding the accuracy, adequacy, validity, reliability, availability or completeness of this report. This report should not be considered or utilized as a complete assessment of the overall utility, security or bugfree status of the code.

This audit report contains confidential information and is only intended for use by the client. Reuse or republication of the audit report other than as authorized by the client is prohibited.

This report is not, nor should it be considered, an “endorsement”, “approval” or “disapproval” of any particular project or team. This report is not, nor should it be considered, an indication of the economics or value of any “product” or “asset” created by any team or project that contracts with Informal to perform a security assessment. This report does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor does it provide any indication of the client’s business, business model or legal compliance. This report should not be used in any way to make decisions around investment or involvement with any particular project. This report in no way provides investment advice, nor should it be leveraged as investment advice of any sort.

Blockchain technology and cryptographic assets in general and by definition present a high level of ongoing risk. Client is responsible for its own due diligence and continuing security in this regard.