Security Audit Report

Stride 2024 Q4 - echosdotfun-contracts

Authors: Darko Deuric, Mahtab Norouzi

Last revised 26 November, 2024

Table of Contents

Audit Overview

The Project

In November 2024, **Informal Systems** conducted a security audit of **Stride's Meme Token Platform**. At its core, the system revolves around the `MemeTokenManager`, a contract enabling users to create meme tokens using native tokens or ERC20 tokens (currently USDC/wUSDC).

Key functionalities include:

1. **Token trading:** Once created, `MemeToken`s can be traded via a **bonding curve pricing model** (`price = Ae^(B*supply)`) governed by the `BondingCurve` contract. This ensures a dynamic price that increases or decreases as tokens are minted or sold, respectively.
2. **Graduation phase:** After the funding phase, trading transitions to a **Uniswap v3 liquidity pool** seeded with 200M meme tokens and the payment tokens collected during funding (base funding amount + graduation fees).

The audit aimed to evaluate the security and robustness of the platform, focusing on critical areas such as phase transition logic, Uniswap integration, price manipulation prevention, fee distribution, and reentrancy protections. Although code changes during the audit slightly slowed the process, close collaboration and frequent communication with the Stride team ensured a thorough evaluation.

Key Findings

While the system demonstrated a solid architecture and thoughtful design, the audit revealed a few notable findings:

1. **Dependency on `balanceOf` for Internal Calculations**
   The `MemeToken` contract relies on the `balanceOf` function to track payment token progress toward the funding goal. Malicious users could bypass the `buy` function by directly sending payment tokens to the contract, artificially inflating its balance and disrupting internal calculations. This could lead to transaction failures during subsequent purchases or token graduation.
   See Manipulation of paymentToken.balanceOf Could Disrupt Funding Progress and Pool Initialization.
2. **Premature Uniswap Pool Creation**
   Malicious users can create a Uniswap v3 pool for a `MemeToken` before it officially graduates, disrupting the intended creation process by the contract. This could prevent the system from initializing the pool during graduation, leading to transaction failures.
   See Premature Uniswap V3 Pool Creation Blocks Token Graduation.
3. **Incomplete Resolution of Premature Pool Issue**
   While a commit introduced logic to check for existing pools before creation, it did not address aligning the funding goal's payment tokens with the current price in an already existing pool. Potential risks include:
   - Transactions failing if the required USDC amount is insufficient.
   - Partial liquidity mismatches.
   - Stranded USDC in the contract.

   See Unresolved Issues with Premature Pool Creation.

Conclusions

Stride's Meme Token Platform represents a strong and modular design, with a well-thought-out architecture that balances functionality and user experience. The system's `MemeTokenManager` is upgradeable, and the development team demonstrated exceptional attention to detail, particularly in addressing edge cases during bonding curve calculations. Examples include:

- Proper handling of first purchases exceeding the funding goal.
- Ensuring exactly 800M meme tokens are minted at the funding goal.
- Safeguards against underflows in sequential purchases.

These thoughtful implementations, along with a collaborative approach to addressing findings during the audit, reflect Stride’s commitment to building a secure and efficient platform.

Audit Dashboard

Target Summary

- **Type**: Protocol and Implementation
- **Platform**: Solidity
- **Artifacts:**
    - MemeTokenManager.sol
    - MemeToken.sol
    - BondingCurve.sol
    - WrappedUSDC.sol

Auditors

- Darko Deuric
- Mahtab Norouzi

Engagement Summary

- **Dates**: 04 Nov 2024 - 25 Nov 2024
- **Method**: Code Review

Severity Summary

Finding Severity	**#**
Critical	-
High	3
Medium	-
Low	-
Informational	5
Total	**8**

Threats

Threats with ✅ are correct, ❗ indicate informational-level warnings, and ❌ represent threats that led to high-severity findings.

Improper initialization

- The `initializer` modifier is not used to prevent re-initialization. ✅
- The owner isn't set correctly. ✅
- Inherited initialization functions are not used properly. ✅
- There are no checks for zero addresses in `initialize()` . ✅
- There is no constructor with `_disableInitializers()` function. ❗

Access control

- Unauthorized access to admin functions - not all necessary functions are marked with `onlyOwner` . ✅
- Unauthorized withdrawal of funds by owner, developer, creator, or other users. ✅

Improper payments

- Overpayment near funding cap might not return surplus funds. ✅
  - `paymentAmountToReturn` is returned to the user.
- Overpayment is causing trading fees to be higher than expected. ✅
  - `tradingFee` is calculated based on `fundingContributionAmount` , which is capped by `remainingFundingNeeded` if required.
- Meme token doesn't have enough funds to pay for graduation fees . ✅
- Trying to sell more than meme token supply ✅
- Attempting to buy or sell with minimal payment (`1 wei` or equivalent).
- Slippage handling is not correct. ✅
- User can pay less then defined fee. ✅
- User can pay more then defined fee ❗
  - see Lack of Refund for Overpaid Native Token During Token Creation .
- The `tradingFeeBps` could be set above 100% or an unreasonably high value ❗
  - `tradingFeeBps` could be very high, e.g. 8000 (80%)
- Setting fees to zero is probably undesirable. ❗
  - no check for 0 value fees

Meme tokens

- Malicious tokens created by users - could override `transfer` and `transferFrom` ✅
  - It's not possible because the creation of meme tokens is fully controlled by the `MemeTokenManager` .
- Invalid inputs - potential issues with excessive length or "weird" characters. ✅
- Invalid inputs - external links (image, Twitter, Telegram, website) are not validated. ❗
- Large number of created meme tokens might cause DoS or storage issues. ❗
  - see deployedTokens Array May Lead to Large RPC Payloads.
- Meme tokens can be bought / sold with a token different than the specified payment token. ✅
- Minting and burning of meme tokens are not handled properly ✅
  - the `_mint` function is called during buying, while `_burn` is used during selling.

Transition issues FUNDING => TRADING & Funding goal impact:

- Transition can occur too early or too late which is causing some issues. ✅
- Transition never occurs - it's blocked by some failing process (like buy or sell). ❌
- Attackers might send funds directly to the contract to manipulate the state. ❌
- Improper amount of payment tokens left on meme token's account after graduation. ❌
    - For the 3 threats listed above, see Premature Uniswap V3 Pool Creation Blocks Token Graduation , Manipulation of paymentToken.balanceOf Could Disrupt Funding Progress and Pool Initialization and Unresolved Issues with Premature Pool Creation
- Transition for all meme tokens happens simultaneously. ✅
    - not possible since meme tokens are fully independent.
- Transition achieved with the first single buy. ✅
    - `fundingContributionAmount` equals `remainingFundingNeeded` , which is equal to the funding goal in this scenario. The `memeAmount` is determined by the bonding curve based on the funding goal, and should be approximately 800M. If the result is less or greater than 800M, adjustments are made to ensure exactly 800M meme tokens are minted.
- Meme supply is greater then 800M when funding goal is reached ✅
- Meme supply is less than 800M when funding goal is reached ✅

Transition issues TRADING => FUNDING

- Transition is not one-directional and irreversible => trading to finding phase switch possible. ✅
    - protected by `onlyNotGraduated` modifier

Reentrancy

- State changes don't occur before external calls. ✅
    - The `recipient.call` function is invoked within `withdrawPaymentToken` (an internal function). Since `withdrawPaymentToken` is the final function called in the `buy` and `sell` functions, and no state changes occur afterward, this implies there is no risk of reentrancy.
- There are single-function reentrancy risks. ✅
- There are cross-function reentrancy risks. ✅

Bonding curve

- Errors in calculations could lead to incorrect pricing. ✅
- Bonding curve isn't disabled when the funding goal is reached. ✅
- Negative values as results from calculations ✅
- x1 (new meme token supply) ends up being < x0 (current meme token supply) after buy function => underflow when calculating meme tokens amount ✅
    - Edge cases are handled by returning 0 meme tokens in certain scenarios. This effectively prevents any buys with tiny payment amounts provided.
- x1 ends up being > x0 after sell function => underflow when calculating payment tokens amount ✅
    - there is explicit require that x0 (current meme token supply) is greater or equal than x1 ()

Liquidity pools

- Parameters during Uniswap pool creation are not valid(ated). ✅
- Large price deviations between final token value and pool price. ❌
    - see Unresolved Issues with Premature Pool Creation.

WrappedUSDC

- Rounding errors and loss of precision ✓
- overflow risks ✓
- no zero value checks ✓

Fees and gas costs ✓

- Fees can be paid in a token different than the payment token. ✓
- Incremental gas cost for new tokens increases with `deployedTokens` array size. ✓
- Unfair gas costs for users triggering state transitions or other events. ❗
  - Issue: Disproportionate Gas Costs for Final Buyer

Findings

Finding	Severity	Status
Premature Uniswap V3 Pool Creation Blocks Token Graduation	HIGH	RESOLVED
Manipulation of paymentToken.balanceOf Could Disrupt Funding Progress and Pool Initialization	HIGH	RESOLVED
Unresolved Issues with Premature Pool Creation	HIGH	RESOLVED
Disproportionate Gas Costs for Final Buyer	INFORMATIONAL	DISPUTED
Suboptimal Placement of Slippage Check	INFORMATIONAL	RESOLVED
deployedTokens Array May Lead to Large RPC Payloads	INFORMATIONAL	RESOLVED
Suboptimal Usage of depositPaymentToken and withdrawPaymentToken in createToken	INFORMATIONAL	ACKNOWLEDGED
Lack of Refund for Overpaid Native Token During Token Creation	INFORMATIONAL	DISPUTED

Premature Uniswap V3 Pool Creation Blocks Token Graduation

Project	Stride 2024 Q4 - MemeToken System
Type	IMPLEMENTATION
Severity	HIGH
Impact	MEDIUM
Exploitability	HIGH
Status	ACKNOWLEDGED

Involved artifacts

contracts/MemeToken.sol

Description

The `createLiquidityPool()` function in the `MemeToken` contract is responsible for creating a Uniswap v3 pool during the token graduation process. However, it does not check if a pool already exists for the token pair and fee tier. This creates an attack vector where a malicious actor can preemptively create a pool for the token before graduation, potentially blocking the token from successfully completing the graduation process.

Problem Scenarios

- An attacker buys some meme tokens during the funding phase.
- The attacker uses their meme tokens and the corresponding payment token (e.g., USDC) to create a Uniswap v3 pool for the token pair before graduation occurs.
- The attacker could create pools at multiple fee tiers (e.g., 0.05%, 0.3%, 1%) to cover all common fee options.
- When the funding phase ends, the `createLiquidityPool()` function is invoked during graduation to create a Uniswap v3 pool for the token pair and initialize it with the desired price and liquidity.
- If a pool with the same token pair and fee tier already exists, the `createPool()` call will revert, likely causing the transaction and the entire graduation process to fail.
- **Impact:**
  - The token's graduation is blocked, leaving it in an incomplete state.
  - This could disrupt the project's tokenomics.

Recommendation

1. **Check for existing pool:**

- Modify the `createLiquidityPool()` function to query the Uniswap v3 factory using the `getPool()` function.
- If a pool already exists for the token pair and fee tier, act accordingly:
  - Use the existing pool if its parameters align with the intended setup.

  - Revert the transaction with a clear error message if the pool is incompatible.

2. **Restrict token transfers during funding:**
   - Restrict token transfers during the funding phase to prevent attackers from obtaining tokens to create pools.
   - Allow only transfers related to purchases or selling back to the meme token contract.

Unresolved Issues with Premature Pool Creation

Project	Stride 2024 Q4 - MemeToken System
Type	IMPLEMENTATION
Severity	HIGH
Impact	HIGH
Exploitability	MEDIUM
Status	RESOLVED

Involved artifacts

- contracts/MemeToken.sol
- contracts/MemeTokenManager.sol

Description

The commit aimed to address the issue of premature pool creation for MemeTokens by ensuring the contract first checks for the pool's existence before proceeding. However, it does not resolve critical issues related to how the correct amount of USDC (or payment token) should be sent to the pool along with 200M MemeTokens to match the pool's current price. If the pool was already created by an attacker with a different price point, the transaction for `uniswapV3NonfungiblePositionManager.mint(params)` could fail, result in a partial match, or leave some amount of USDC stuck in the MemeToken contract.

Problem Scenarios

- An attacker prematurely creates a pool for the MemeToken using USDC, setting an arbitrary price.
- When the token graduates, the contract tries to add 200M MemeTokens and the amount of USDC equal to `fundingGoal - graduation fees`.
- If the attacker's price in the pre-existing pool differs significantly from the final bonding curve price:
  - The transaction to add liquidity might fail entirely.
  - Partial matching could occur, leaving leftover USDC on the MemeToken contract.
  - The available USDC (`fundingGoal - graduation fees`) might be insufficient to match the pool price, causing the transaction to fail.

Recommendation

Adopt the Stride team's suggestion to create an **empty pool** immediately after the MemeToken is created. This ensures that no other user can create a conflicting pool afterward. The contract can initialize the pool with the correct liquidity amounts and prices during graduation. Restrict transfers **to the pool address** until the graduation phase, allowing only legitimate buys/sells within the MemeToken contract. Other transfers, including user-to-user, remain unrestricted. This restriction is lifted once the pool is initialized during graduation.

Manipulation of paymentToken.balanceOf Could Disrupt Funding Progress and Pool Initialization

Project	Stride 2024 Q4 - MemeToken System
Type	IMPLEMENTATION
Severity	HIGH
Impact	HIGH
Exploitability	MEDIUM
Status	RESOLVED

Involved artifacts

- contracts/MemeToken.sol
- contracts/MemeTokenManager.sol

Description

The `paymentToken.balanceOf` function is used in several critical parts of the `MemeToken` contract, such as tracking progress toward the funding goal and determining the initial price / initial amount of liquidity pool. However, an attacker can directly transfer `paymentToken` (e.g., USDC) to the `MemeToken` contract, bypassing the `buy` function. This unintended behavior can artificially inflate the contract's balance, disrupting the internal logic and potentially halting token operations.

Problem Scenarios

- **Setup:**
  - `fundingGoal()` is set to 10,000 USDC.
  - The current `paymentToken.balanceOf(MemeToken)` is 9,800 USDC, reflecting legitimate buys and sells.
- **Attack:**
  - An attacker directly transfers 300 USDC to the `MemeToken` contract, bypassing the `buy` function.
  - The new `paymentToken.balanceOf(MemeToken)` becomes 10,100 USDC.
- **Impact on buy function:**
  - When a legitimate user calls `buy`, the `quoteBuy` function calculates `remainingFundingNeeded` as:

    ```
    int256 remainingFundingNeeded = fundingGoal() -
    paymentToken.balanceOf(address(memeToken));
    ```

    With the inflated balance, `remainingFundingNeeded = 10,000 - 10,100`, which results in an **underflow** due to Solidity's safe math checks (version ≥0.8).
  - This causes the **transaction to revert**, effectively blocking all future `buy` calls.
- **Impact on pool initialization:**
  - During graduation, `paymentToken.balanceOf(memeToken)` is used to determine `amount1` (the amount of payment token) for liquidity pool creation.
  - The artificially inflated balance leads to an incorrect initial price in the Uniswap V3 pool, disrupting the intended price alignment with the bonding curve.

Relying on `paymentToken.balanceOf` as the sole source of truth for the funding progress and pool initialization exposes the contract to manipulation. Direct transfers to the contract bypass the intended logic in the `buy` function and can artificially inflate the balance.

Recommendation

- Create an internal variable (e.g., `collectedFunds`) to track the total payment tokens received via the `buy` function.
- Increment this variable only through the `buy` function and decrement it through the `sell` function.
- Use this variable to calculate funding progress instead of relying on `paymentToken.balanceOf`.
- During graduation, use `collectedFunds` (potentially reduced by graduation fees) instead of `paymentToken.balanceOf` to determine the amount of payment tokens to add to the liquidity pool. This ensures the pool's initial price is aligned with the bonding curve.

Disproportionate Gas Costs for Final Buyer

Project	Stride 2024 Q4 - MemeToken System
Type	IMPLEMENTATION
Severity	INFORMATIONAL
Impact	NONE
Exploitability	HIGH
Status	DISPUTED

Involved artifacts

contracts/MemeToken.sol

Description

The user who triggers the transition to the "trading" stage by calling `buy()` ends up paying gas fees for all the actions within the `if funding goal is reached` block. This results in potentially higher gas costs for this final buyer compared to previous buyers who did not trigger this logic.

Problem Scenarios

- User N triggers the transition to "trading" by calling `buy()`.
- The following actions are executed, incurring additional gas costs:
  - Sending a graduation reward to the creator.
  - Sending a graduation fee to the developer.
  - Creating a Uniswap v3 liquidity pool and adding initial liquidity.
  - Updating the contract's stage to "trading."
- The additional gas costs are borne entirely by the final buyer, potentially making this transaction disproportionately expensive.

For rollup-based chains, these additional gas costs may be negligible. However, it would be beneficial to analyze the potential gas impact further to determine the best solution.

Recommendation

- Consider splitting the transition logic into a separate admin or system-triggered function. This allows the final buyer to avoid incurring additional gas costs.
- Alternatively, explore ways to reduce gas usage within the `if` block or implement mechanisms to compensate the final buyer, such as offering a slight token discount.

Suboptimal Placement of Slippage Check

Project	Stride 2024 Q4 - MemeToken System
Type	IMPLEMENTATION
Severity	INFORMATIONAL
Impact	NONE
Exploitability	LOW
Status	RESOLVED

Involved artifacts

contracts/MemeToken.sol

Description

The slippage check in the `buy()` function is currently placed **after** `paymentToken` transfers. This design leads to unnecessary gas expenditure if the transaction later reverts due to the slippage check failure.

Problem Scenarios

- The `paymentToken.transferFrom` and `paymentToken.transfer` calls are executed before the slippage check.
- If the slippage check fails (`memeAmount < minReceived`), the transaction reverts **after** the payment tokens have been transferred, resulting in wasted gas for the user.

Recommendation

Move the slippage check **before** the `paymentToken` transfers to:

1. Align better with the **check-effects-interactions** pattern.
2. Save gas in cases where the transaction would revert.

deployedTokens Array May Lead to Large RPC Payloads

Project	Stride 2024 Q4 - MemeToken System
Type	IMPLEMENTATION
Severity	INFORMATIONAL
Impact	NONE
Exploitability	MEDIUM
Status	RESOLVED

Involved artifacts

contracts/MemeTokenManager.sol

Description

The `deployedTokens` array in the `MemeTokenManager` contract, while only used for off-chain purposes through the `getDeployedTokens()` getter function, could lead to increased burden on RPCs when queried as the array grows. This is due to the large payload size being sent back, especially in scenarios with a significant number of deployed tokens.

Problem Scenarios

- The `deployedTokens` array stores addresses of all deployed `MemeToken` instances.
- Nodes querying this array through `getDeployedTokens()` may experience delays or inefficiencies due to large payload sizes, particularly as the number of tokens increases.
- While gas costs are not impacted (since the getter is view-only), the potential strain on RPCs and inefficiencies in returning large datasets could become problematic over time.

Recommendation

- **Implement pagination** in the getter function to limit the size of the data being sent in each response. This ensures efficient handling of large datasets by RPCs and reduces the likelihood of performance issues.
- Alternatively, consider **removing the array** and relying solely on emitted events to track deployed tokens, as they are already indexed and queryable off-chain.

Suboptimal Usage of depositPaymentToken and withdrawPaymentToken in createToken

Project	Stride 2024 Q4 - MemeToken System
Type	IMPLEMENTATION
Severity	INFORMATIONAL
Impact	NONE
Exploitability	MEDIUM
Status	ACKNOWLEDGED

Involved artifacts

contracts/MemeTokenManager.sol

Description

The `createToken` function currently uses `depositPaymentToken` and `withdrawPaymentToken` to handle developer fee payments when `useNativeTokenForPayment` is `true`. This results in wrapping Ether into WETH and then immediately unwrapping it back into Ether to transfer to the developer. This process is potentially unnecessary and suboptimal.

Problem Scenarios

- Wrapping and unwrapping involve external calls to the WETH contract, consuming additional gas.
- The wrapping-unwrapping cycle adds redundant steps to the transaction without providing any functional benefits.

Recommendation

Directly transferring Ether to the developer would be more efficient and simpler in this specific scenario.

Lack of Refund for Overpaid Native Token During Token Creation

Project	Stride 2024 Q4 - MemeToken System
Type	IMPLEMENTATION
Severity	INFORMATIONAL
Impact	NONE
Exploitability	LOW
Status	DISPUTED

Involved artifacts

contracts/MemeTokenManager.sol

Description

In the `createToken` function, if `msg.value > creationDeveloperFeeAmount`, the excess native tokens (ETH) paid by the user are not refunded. This issue occurs because the function does not implement a mechanism to return overpaid amounts.

Problem Scenarios

- A user calls the `createToken` function with `msg.value` exceeding `creationDeveloperFeeAmount`.
- The function processes the required amount (`creationDeveloperFeeAmount`) but leaves the excess amount unhandled.
- This results in the user losing any overpayment without a refund.
  The team mentioned that implementing a refund check at the contract level is not currently feasible due to compatibility issues with the `createAndBuyToken` function, which relies on combining payments for creation and buying.

Recommendation

While the team plans to rely on UI-level enforcement to prevent overpayment, consider implementing an event that logs overpaid amounts for better transparency. If feasible in the future, explore adding a refund mechanism without affecting the `createAndBuyToken` function, ensuring robustness for users interacting directly with the contract.

Disclaimer

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**
🟠 **High**	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
🟡 **Medium**	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)
🟢 **Low**	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)
🔵 **None**	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:

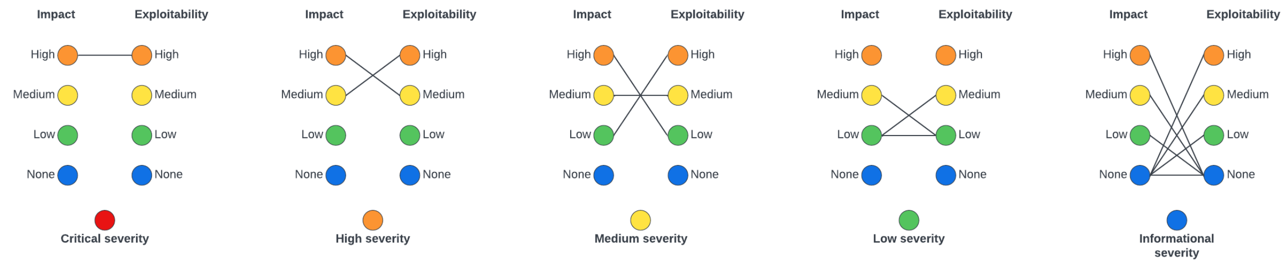
- *Actors* can be any entity that interacts with the system: other blockchains, system users, validators, relayers, but also uncontrollable phenomena (e.g. network delays or partitions).
- *Actions* can be

    - *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**
🟠 **High**	illegitimate actions taken by a small group of actors; possibly coordinated with legitimate actions taken by a medium group of actors
🟡 **Medium**	illegitimate actions taken by a medium group of actors; possibly coordinated with legitimate actions taken by a large group of actors
🟢 **Low**	illegitimate actions taken by a large group of actors; possibly coordinated with legitimate actions taken by all actors
🔵 **None**	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**
Severity Score	**Examples**
🔴 **Critical**	Halting of chain via a submission of a specially crafted transaction

Severity Score	Examples
🟠 **High**	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
🟡 **Medium**	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
🟢 **Low**	2x increase in node computational requirements via coordinated withdrawal of all user tokens
🔵 **Informational**	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