Security Audit Report

dYdX 2024 Q1: Risk&Safety, Liquidation daemon

Authors: Andrija Mitrovic, Mirel Dalcekovic

Last revised 7 May, 2024

Table of Contents

Audit Overview

The Project

In March 2024., dYdX has engaged Informal Systems to work on partnership and conduct security audit of the following items in dYdX's open source software:

1. Risk and Safety: Subaccounts Withdrawal Gating
2. Risk and Safety: Withdrawal speed limits and rate limits
3. Liquidation Daemon improvements

Scope of this report

The agreed-upon work plan consisted of the following tasks:

- Analyze implemented improvements and safety mechanisms:
    - Withdrawal speed limits, rate limits:
        - Are there ways to bypass the rate limits?
    - Subaccounts withdrawal gating:
        - Are there any ways to bypass withdrawal halts?
- Liquidation daemon refactoring:
    - Did this refactoring miss something?
    - Did it do what it was intended?
    - Are there any negative TNCs liquidatable subaccounts missing?
    - Is it doing incorrect collateralization checks?
    - Performance improvements - are there any ways to make them better?

Audit plan

The audit was conducted between March 4th and April 4th, 2024.

Timeline

- 05.03.2024. Kick off meeting, featuring scope agreement and a high-level code walkthrough conducted by the dYdX team for the Withdrawal gating, Rate limits and Liquidations daemon refactoring.
- 19.03.2024. Weekly sync meeting and discussion about v3 incident with dYdX team representatives.
- 20.03.2024. Discussion with team representatives about liquidations and withdrawal halts. First findings shared on the meeting.
- 26.03.2024. Weekly sync meeting. Sharing progress with dYdX team representatives.
- 29.03.2024. Draft report shared
- 29.03.2024. Final report for 2023 Q4 shared.
- 04.04.2024. Closure meeting
- 05.04.2024. Final report for 2024 Q1 shared.
- 07.05.2024. New revision of final report for 2024 Q1 audit restitution. Updated report to incorporate minor dYdX Trading legal team revisions for clarity and accuracy.

Conclusions

Upon conducting a thorough examination of the project, it was noted that the audited new features and improvements do not entail a substantial amount of changes in the code base. However, they do reveal several protocol-level flaws, as outlined in the Findings section.

Audit Dashboard

Target Summary

- **Type**: Protocol and Implementation
- **Platform**: Go
- **Artifacts** audited over protocol/v4.0.0-rc3 branch and commit hash 5d118b1
    - Risk and Safety projects:
        - Global Withdrawal Speed Limits
        - Subaccount Withdrawal Gating
    - Liquidations daemon refactoring

Engagement Summary

- **Dates**: 04.03.2024. - 04.04.2024.
- **Method**: Manual code review, protocol analysis

Severity Summary

Finding Severity	**#**
Critical	0
High	0
Medium	2
Low	1
Informational	3
Total	6

Findings

Title	Type	Severity	Status	Issue
Proposed operations tx has no size limitation leading to possible withdrawal halting being postponed	IMPLEMENTATION	MEDIUM	ACKNOWLEDGED	
Withdrawal halting could be postponed as long as malicious validators and daemons are selected for the proposer consecutively	PROTOCOL	MEDIUM	ACKNOWLEDGED	
Ensure WITHDRAWAL_AND_TRANSFERS_BLOCKED_AFTER_CHAIN_OUTAGE_DURATION chain outage is being observed	PROTOCOL	LOW	ACKNOWLEDGED	
Suggestion to move rate limiting protocol to the Prepare Proposal	PROTOCOL	INFORMATIONAL	ACKNOWLEDGED	
An attacker can misuse the lack of upper limit on capacity to increase it using timeout/failed packets	PROTOCOL / IMPLEMENTATION	INFORMATIONAL	ACKNOWLEDGED	
Change implementation and handling of protocol-defined invariants	PROTOCOL / IMPLEMENTATION	INFORMATIONAL	ACKNOWLEDGED	

Proposed operations tx has no size limitation leading to possible withdrawal halting being postponed

Project	**dYdX 2024 Q1: Risk&Safety, Liquidation daemon**
Type	IMPLEMENTATION
Severity	MEDIUM
Impact	HIGH
Exploitability	LOW
Status	ACKNOWLEDGED
Issue	

Involved artifacts

- /protocol/app/prepare/prepare_proposal.go

Description

Sentinel deleverage operation with zero amount fills is added as an additional deleveraging operation during the `PrepareCheckState`, after the liquidations and deleveraging are finalized if there are still negative TNC (Total Net Collateral) accounts present.
All the prepared operations (order matches, liquidation matches and deleveraging matches with order removals) are being injected to the proposed block by the proposer within one operations tx.
Currently, there are no size checks implemented for this transaction, raising the possibility that the prepared block might surpass limitations and consequently be rejected, prompting the consensus to shift to the next proposer.

In scenarios where the transaction size is primarily influenced by the quantity of liquidations and required deleveraging, we may encounter liveness issues. This arises from the deterministic nature of the process, as all validator nodes would (if honest) report the same set of subaccounts for liquidation and those with negative TNC.

The order of filling the proposed block available `tx` bytes are as following:

1. First, the fixed group size transactions are placed (price updates, premium votes and acknowledgment for the bridged txs)
2. After, the remaining bytes are calculated and 25% are allocated for the “other group” transactions - here
3. After this point, proposed operations are placed in the operations txs - here
4. If any additional room is left, it is filled with “other group” transactions - here

and after all the `txs` are inserted, they are being reordered before returning the tx to the CometBFT:

1. proposed operations tx
2. other txs
3. ack bridges tx

4. funding samples/premium votes tx
5. price updates tx

This ensures that there will be no inconsistency when processing other tx if block is marked as one containing the negative TNC account.

Problem Scenarios

In case that proposed operations `tx` is to big to fit the available bytes for the block size:

- Protocol would not be informed that there is a negative TNC account present and the desired withdrawal halt would be postponed. We see this as critical issue.
- If size limitation were introduced - what would be the logic for adding proposed operations - all of them/ some - they are dependable between themselves.

Recommendation

Introduce constraints on the proposed transaction size for operations.

Implement additional logic to guarantee that proposed operations including the sentinel deleveraging operation are included in the transaction. One potential solution could involve bypassing the addition of other group transactions to ensure that the operations category transaction is placed in the block, or at the very least, including the sentinel deleverage operation in the operations transaction.

These suggestions for implementation and design changes should be thoroughly considered before implementation.

Following discussions with dYdX team members, it has been conveyed that although the only live mainnet deployment of the software is not currently approaching the maximum block size, mainnet deployers of the software should introduce monitoring of block size (if not already in place) and take the appropriate actions if block sizes begin to increase.

Withdrawal halting could be postponed as long as malicious validators and daemons are selected for the proposer consecutively

Project	**dYdX 2024 Q1: Risk&Safety, Liquidation daemon**
Type	PROTOCOL
Severity	MEDIUM
Impact	HIGH
Exploitability	LOW
Status	ACKNOWLEDGED
Issue	

Involved artifacts

- liquidations protocol design

Description

The validators are running their own liquidations daemon and providing the information about liquidatable subaccounts in the system. In case they are malicious, they could skip reporting the most underwater subaccounts to the on-chain logic to process liquidation.

If there are more than one malicious validators and liquidation daemons in the system, they could collaborate on this.

Problem Scenarios

There is no mechanism to detect whether or not the liquidation daemons reported all they should have.

In context of the analyzed withdrawal subaccounts, this could lead to postponing the withdrawal halts as long as malicious validators are selected for the proposers:

- updates of subaccount will be blocked only in case of updating to "more risky" state
- all other updates will be allowed

Recommendation

Set up the monitoring of the liquidation daemons work - track nodes that are missing to report subaccounts comparing to the majority. Think of changing the design with introducing VE and injecting the liquidatable subaccounts with vote extensions.

Ensure WITHDRAWAL_AND_TRANSFERS_BLOCKED_AFTER_CHAIN_OUTAGE_DURATION chain outage is being observed

Project	**dYdX 2024 Q1: Risk&Safety, Liquidation daemon**
Type	PROTOCOL
Severity	LOW
Impact	LOW
Exploitability	LOW
Status	ACKNOWLEDGED
Issue	

Involved artifacts

- /protocol/x/subaccounts/keeper/subaccount.go
- /protocol/x/blocktime/keeper/keeper.go

Description

If no period that lasted for `WITHDRAWAL_AND_TRANSFERS_BLOCKED_AFTER_CHAIN_OUTAGE_DURATION` is observed, the `GetDowntimeInfoFor` function will return largest period observed data, smaller than the `WITHDRAWAL_AND_TRANSFERS_BLOCKED_AFTER_CHAIN_OUTAGE_DURATION` value.

`internalCanUpdateSubaccounts` function contains check if the `chainOutageExists` :

```
downtimeInfo := k.blocktimeKeeper.GetDowntimeInfoFor(
        ctx,
        types.WITHDRAWAL_AND_TRANSFERS_BLOCKED_AFTER_CHAIN_OUTAGE_DURATION,
)
chainOutageExists := downtimeInfo.BlockInfo.Height > 0 && downtimeInfo.Duration > 0
    ...
}
```

Problem Scenarios

The above described behavior would lead to withdrawal halts in case of shorter outages than predefined with the implementation and the documentation.

Recommendation

Suggestions:

- Additional validations ensuring that Downtime parameters include `WITHDRAWAL_AND_TRANSFERS_BLOCKED_AFTER_CHAIN_OUTAGE_DURATION` period are needed or
- upon getting `downTimeInfo` here in `internalCanUpdateSubaccount` function add additional condition and check if the `downtimeInfo.Duration == WITHDRAWAL_AND_TRANSFERS_BLOCKED_AFTER_CHAIN_OUTAGE_DURATION`.

After discussions with the dYdX team regarding this finding, it was determined that due to the pessimistic logic used in the `x/blocktime` module's outage periods observation, this feature aligns with expectations. If the exact period is not being observed and if a chain outage is possible to occur, it will be considered as having happened. This leads to the possibility of halting being triggered for outages that are shorter or have not occurred.

Based on these expectations, the impact of this issue is reduced, contributing to an overall severity rating of Low.

The existing withdrawal subaccounts gating specifications should be clarified and aligned with this implementation. The dYdX team has acknowledged that validations ensuring the exact chain outage duration is observed with the `x/blocktime` module should be introduced.

Suggestion to move rate limiting protocol to the Prepare Proposal

Project	**dYdX 2024 Q1: Risk&Safety, Liquidation daemon**
Type	PROTOCOL
Severity	INFORMATIONAL
Impact	UNKNOWN
Exploitability	UNKNOWN
Status	ACKNOWLEDGED
Issue	

Involved artifacts

- /protocol/x/ratelimit/keeper/keeper.go

Description

This is not an issue, this is just a suggestion to be considered.

The current implementation of the withdrawal rate limiting protocol means that each transaction (deposit, withdraw, acknowledgement and timeout) will be processed trough `DeliverTx` in order that it is put in the block and thus changing the capacity. No reordering of these transaction is possible. Without reordering, it can happen that some withdrawal gets rejected because their amount is larger than the current capacity but then, right after this rejection, comes a deposit (or a failed/timeout transaction) that will increase the capacity so that the previously rejected transaction could have passed.
Reordering of the transactions can be done in `PrepareProposal` by the proposer, allowing it to select the sequence of deposit and withdrawal transactions. This can be done in a way to maximize the usage of capacity in one block by refilling the capacity first (by deposits or failed or timeout packets) to make it possible for the incoming withdrawal transactions to pass.

Problem Scenarios

One such scenario where the rate limiting protocol in `PrepareProposal` has an upside is the following:

Lets assume that there are two transactions to be processed in the current block. One is a withdrawal (`withdrawalAmount = 500`) and the other one is a deposit (`depositAmount = 300`). Assume that capacity is 400 at the beginning of the block. Scenario:

1. Withdrawal happens first and it is rejected because `withdrawalAmount > capacity (500 > 400)`
2. Deposit happens and capacity gets increased (`capacity = 400 + 300 = 700`)

It can be seen that if the deposit happened first the withdrawal would have passed. Proposer can order these transactions in a Prepare proposal, thus making it possible for the withdrawal tx to pass. This can be more generalized, thus maximizing the number of processed withdrawals per block.

Recommendation

Consider potential upsides of moving withdrawal rate limiting protocol to the `PrepareProposal`. At the same time this suggestion should be thoroughly analyzed because this is yet another protocol to be added to `PrepareProposal` that must be deterministic and also increases complexity and duration of `PrepareProposal`.

An attacker can misuse the lack of upper limit on capacity to increase it using timeout/failed packets

Project	**dYdX 2024 Q1: Risk&Safety, Liquidation daemon**
Type	PROTOCOL IMPLEMENTATION
Severity	INFORMATIONAL
Impact	HIGH
Exploitability	NONE
Status	ACKNOWLEDGED
Issue	

Involved artifacts

- /protocol/x/ratelimit/keeper/keeper.go

Description

Rate limiting capacity value gets increased each time that failed acknowledgement or a timeout packet is received. Increase of capacity happens by the amount that was initially sent and failed or reached a timeout. This increase does not have any upper boundary. The lack of upper boundary can enable a malicious user to withdraw any amount if enough failed and timeout packages get processed before his withdrawal in the same block (or even in the previous blocks because the decrease is amortized in the end block).

Problem Scenarios

Assume that we have a malicious user that wants to withdraw amount S that is larger than the current capacity C. This malicious user can misuse failed or timeout packets to increase the current capacity on the source chain (dYdX).

There are two possible ways that this can happen:

1. Unintentional increase of capacity (by pure chance)
2. Intentional increase of capacity at a specific height (time)

Unintentional increase of capacity (by pure chance)

Assumptions:

- There are many packages that failed or timed out, waiting to be processed on the source (dYdX) chain. Enough of these packages, combined with the current capacity, add up to more than the sum S that the malicious user wants to withdraw.

Scenario:

1. Malicious user submits withdrawal transaction.
2. This transaction ends up to be processed in a block after failed or timeout transactions.
3. Each failed or timed-out transaction is processed individually, with each one increasing the current capacity, C. This ensures that C reaches a value equal to or greater than the withdrawal amount, S.
4. The withdrawal transaction submitted by the malicious user is processed and passes the capacity check because the withdrawal amount, S, is less than or equal to the current capacity, C.

Intentional increase of capacity at a specific height (time)

Assumptions:

- Attacker has a significant amount of funds on one or multiple accounts (almost the double amount that he wants to withdraw).
- There is a connected destination chain to dYdX that doesn't execute incoming packets, causing them to reach their timeouts without being processed.

Scenario:

1. Attacker divides the amount S in several sub-amounts that are smaller than the current capacity, C.
2. Attacker creates withdrawal packets for each of these amounts and creates them in a way that they will not be executed on the destination chain. He also sets their timeout for some distant block height (same for all packets) on the destination chain or chains. He does this in order to give time to the rate limiting protocol on dYdX to restore the capacity after each packet is sent but making all the packets timeout at the same time/ block height.
3. Just before the timeout is reached attacker creates a withdrawal packet with the desired amount S that is still larger than the capacity, C
4. If these timeouts are received and processed in the same block (or even in some previous blocks because the decrease of capacity is amortized in the end block) as the new withdrawal packet but before him the capacity will be increased enough thus leaving the possibility of withdrawing the initial amount S, much larger that the initial current capacity C.

In both of these scenarios a malicious user has ended up with an increased capacity that enabled him to withdraw the desired amount S.

Recommendation

We recommend introducing an upper limit to the value of capacity. This will prevent intentional or unintentional increase in capacity over some value.

After discussions with the dYdX team regarding this finding, it was concluded that the scenarios explained with this issue do not show any justified reason for a malicious user to act in order to increase the capacity and then pull a larger amount than initially possible. This is acceptable to the dYdX team - they feel there is no reason or motivation to exploit this issue.

Based on this assessment, the exploitability of this issue was lowered to None, contributing to an overall severity rating of Informational.

Change implementation and handling of protocol-defined invariants

Project	**dYdX 2024 Q1: Risk&Safety, Liquidation daemon**
Type	PROTOCOL IMPLEMENTATION
Severity	INFORMATIONAL
Impact	NONE
Exploitability	NONE
Status	ACKNOWLEDGED
Issue	

Involved artifacts

- /protocol/x/ratelimit/keeper/keeper.go
- /protocol/x/ratelimit/util/capacity.go

Description

Within the rate limiting feature, there are two invariant checks performed:

1. Number of capacity data and number of the limiters defined for the denom should be the same, since capacities exist per limiter:
   `len(limiters) == len(capacity_list)` - here
2. `x/blocktime` module keeps information about previous block time and this value should always be less than current block's time, ensuring that:
   `current block time > prev block time` - here

If these invariants are broken, the system proceeds with its execution and simply logs the error: here and here.

Problem Scenarios

An invariant represents a protocol's property that must consistently hold true.

Any violated invariant should have a corresponding response from the application, most appropriate is to halt the chain. This proactive measure prevents further, potentially more problematic consequences arising from the protocol operating in an unforeseen state.

Currently, existing checks are executed from the `x/ratelimit` module's `EndBlocker` and are only logging the error in case of broken invariants, allowing the application to continue running and create potentially bigger issues.

Recommendation

The Cosmos SDK provides comprehensive documentation outlining a recommended approach for defining and managing application invariants. This documentation offers detailed explanations accompanied by illustrative examples to facilitate understanding and implementation.

We would suggest:

1. Define the invariant to ensure **consistency between the number of capacities data and limiters** for the `x/ratelimit` module. This invariant can be verified:
    a. Explicitly, by invoking the message subsequent to chain upgrades or governance proposals that modify the Limit parameters within the system.
    b. Periodically, perhaps on an epoch basis. It may not be necessary to check it at every block level, as the number of data stored in the state typically does not change frequently.
2. Define the invariant to ensure the **previous block time is smaller than current block time** check for the `x/blocktime` module. This invariant can be verified in the `x/blocktime` `BeginBlocker` and if it is broken, application should act accordingly.

With explicit call of `AssertInvariants` per module, special logic defined for each broken invariant should be executed similar to shown in the Cosmos SDK crisis module - here.

As we didn't identify any external actions causing these invariants to break, we're reporting this finding as having None Exploitability. However, the current implementation is flawed from both protocol and application design perspectives.

After discussions with the dYdX team regarding this finding, it was shared that the error-log-only approach was an explicit decision on the behavior of the software. Core trading activities should not be halted when `x/ratelimit` can't function as expected. It's considered a best-effort protection of funds from leaving the chain, but the liveness of the chain is valued more than this protection.

Based on this assessment, the impact of this issue was lowered to None, while the severity remained Informational. Still, the suggestion for defining invariants across the system as explained in the Cosmos SDK documentation stands.

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:

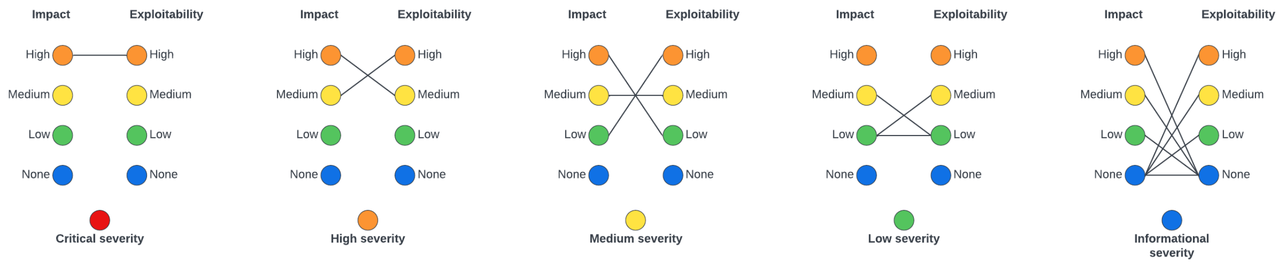
- *Actors* can be any entity that interacts with the system: other blockchains, system users, validators, relayers, but also uncontrollable phenomena (e.g. network delays or partitions).
- *Actions* can be

    - *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**
🔴 **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

Disclaimer