Security Audit Report

Neutron Q3 2025: DEX Fractional Banking Audit

Last Revised
2025/07/30

Authors:
Ivan Gavran, Aleksandar Stojanovic

Contents

Audit overview

The project

In July 2025, Neutron engaged Informal Systems↗ to conduct a security audit of the new feature, Fractional Banking, captured in PR#938↗.

The auditors joined the work before the PR was fully finalized, thus the audit proceeded in two stages:

1. Initial review and comments on commit `2a76a47`↗
2. Follow-up analysis on commit `2283112`↗.

Scope of this report

The scope of the report was analyzing the fractional banking feature as well as its consequences on the functioning of the whole DEX.

Audit plan

The audit was conducted between July 7th, 2025, and July 16th, totalling to 8 person-days 2025 by the following personnel:

- Ivan Gavran
- Aleksandar Stojanovic

Conclusions

We performed the audit combining manual code review and writing property-based tests. We found that overall the feature did not introduce serious adversarial consequences. We found some problems in the implementation, reducing precision or enabling DoS attacks (more details can be found in the "Findings" section). Our findings were promptly patched and reaudited.

Audit Dashboard

Severity Summary

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

System Overview

Initially, the Neutron DEX operated using standard Cosmos SDK 18-digit decimal precision, which was adequate for manual trading where occasional rounding errors were tolerable, but became problematic as automated strategies emerged.

The motivation for implementing high-precision calculations stems from the DEX's architecture supporting automated market making strategies. With liquidity providers able to place positions across numerous ticks (potentially every tick), small rounding errors from standard 18-digit-precision could accumulate rapidly, leading to significant value leakage over time.

The DEX now uses 27-digit precision (10ˆ27) for all internal calculations while maintaining API compatibility. Users continue to interact with the system using standard integer token amounts, but internally:

1. All DEX operations are performed using `PrecDec` with 27-digit precision
2. Fractional balances are tracked separately in the `KVStore` via the `FractionalBanker`
3. Automatic conversion occurs when fractional amounts accumulate to whole tokens, which are then transferred to users' standard bank balances

This approach prevents precision loss while ensuring that no user value is lost to rounding errors.

Threat Model

Here we list the desired properties of the system. We used those properties to analyze what are all possible ways in which they could be violated.

Definitions

For an individual interaction with the system in which the user is getting funds from the system, and for a single chosen coin, let

- `single_I` (single transaction `I`deal calculation) be the amount the user should get [give] if coins could be given in an arbitrary precision. (The full expression is `single_I(c, tx, u)`, for a coin `c`, transaction `tx`, and user `u`)
- `single_R` (single transaction `R`eal calculation) be the amount the user really gets [gives] (full: `single_R(c, tx, u)`)

Similarly, let

- `total_I` be the total amount the user should get [give] in all transactions by now. (The full expression is `total_I(c, t, u, [tx])`, for coin `c` , timepoint `t` , user `u`, and a sequence of transactions `[tx]`.)
- `total_R` be the total amount the user gets [gives] in all transactions by now. (Full: `total_R(c, t, u, [tx])`)

Note, `total_I` and `total_R` get updated in two ways:

- when the user gets some amount (using `SendFractionalCoinsFromModuleToAccount`): contributes a positive amount.
- when the user gives some amount (using`SendFractionalCoinsFromAccountToModule`): contributes a negative amount.

Properties

1. [SINGLE-TX-DRIFT] For each transaction`-1 < single_I - single_R < 1`.
2. [USER-AMOUNT-RESPECTED] If the user is ready to deposit `x` amount of a coin, the exchange will not ask to transfer `>x`.
3. [TOTAL-DRIFT] For the total withdrawn amount, at any point in time, `0 <= total_I - total_R < 1` . In particular, this implies that the DEX is at no point in time at a loss with respect to any of its users.
4. [REMAINDER-CONSISTENCY] For each coin and user, the fractional amount stored for them stores the amount that the DEX owes them. In particular, for `x` being the fractional amount stored, it holds:
    1. Once the next withdrawing transaction takes place, `x = total_I - total_R`
    2. After each withdrawing transaction, `0<=x<1`
    3. At any point, `x >= 0`

    Note 1: [REMAINDER-CONSISTENCY] property implies eventual 1-off fairness: namely, that users can always enforce getting their tokens back (with the tolerance of 1 micro token)

    Note 2: [REMAINDER-CONSISTENCY] looks very similar to [TOTAL-DRIFT]. However, it contains added restrictions (a withdraw needs to happen) because the analysis showed that [TOTAL-DRIFT] did not hold, so

[REMAINDER-CONSISTENCY] was sketched as a replacement property.

5. [EXCHANGE-POSITIVE] Let `S` be the sum total amount on the exchange (for a particular coin). Let `F` be the sum total stored under fractional remainders. Finally, let `Claims` be the sum total of all possible claims (`Withdraw` and `WithdrawFilledLimitOrder`). At all time it holds that `S - F >= Claims`.
6. [NO DUST POSITIONS] At all times, no open position exists with a balance strictly smaller than `1 uToken`
7. [INTERNAL-PRECISION] All calculations are done in `PrecDec`, and the conversion only happens when interfacing with external accounts.
8. [BEHAVIOR-PRESERVATION] If a user was able for a transaction `tx` get value `x` before introducing fractional accounting, not (after introducing it) the user gets `>=x`.

Threat Analysis

[SINGLE-TX-DRIFT] For each transaction `-1 < single_I - single_R < 1`**.** `OK`, `NOT OK`

The property does not hold in its entirety since `single_I - single_R` can fall unboundedly due to the fact that `SendFractionalCoinsFromAccountToModule` will not take into account any debt, and will return it only eventually, when the user will be withdrawing for the first time (thus accummulating the debt, which causes `single_I` to grow, whereas `single_R` remains small).

Let us focus on only those transactions where the user is getting coins from the module (`single_I, single_R >= 0`).

The only sending (except for shares coins, which is the issue already raised) happens in the function `SendFractionalCoinsFromModuleToAccount`.

- Fractional and Whole amounts are properly separated, zeros are not included.
- Whole amounts are sent to the `address` account.
- Fractional parts are stored using `SetFractionalBalance`.

[USER-AMOUNT-RESPECTED] If the user is ready to deposit `x` **amount of a coin, the exchange will not ask to transfer** `>x`**.** `OK`

Sending with rounding up↗ will always result in a number smaller than the initial number of coins the user wanted to send.

Sending with rounding up is done at three places: at `PlaceLimitOrder`, `MultiHopSwap`, and `Deposit`.

1. Let's start by analyzing `PlaceLimitOrder`. When sending↗ coins, it needs to hold that `totalInCoin.amount <= amountIn`. This corresponds to the requirement towards the Swap↗ function that `inAmount <= maxAmountTakerIn`.
2. `LimitOrderTranche.Swap`: `inAmount` is explicitly capped↗ by `maxAmountTakerIn`.
3. `Pool.Swap`: `amountTakerIn` (corresponding to `inAmount`) is explicitly capped↗ by `maxAmountTakerIn`.
4. In `MultiHopSwap`, the module sends exactly↗ the user's number of coins `initialInCoin`.
5. In `ExecuteDeposit`, we have to make sure that `SUM(amounts0) < totalInAMount0` and `SUM(amounts1) < totalInAMount1`. If there is `SwapOnDeposit`, for a deposit that is *behind-enemy-lines,* a token may be reduced a bit when calling `PerformSwapOnDepositSwap`, but cannot increase. If autoswap was disabled, again `inAmount0` or `inAmount1` may be reduced (through the call to `GreatesMatchingRatio`. If it were enabled, the value is unchanged. In the end, `totalInAmount0` (and symmetrically for `totalInAmount1`) is calculated by summing up all those potentially only reduced values.

[TOTAL-DRIFT] At any point in time, `0 <= total_I - total_R < 1`**.** `OK`, `NOT OK`

Does not hold fully, for the same reasons as [SINGLE-TX-DRIFT].

[REMAINDER-CONSISTENCY] `OK`

At each send (from the module to an account, and from an account to the module), the fractional part of the `PrecDec` number is stored, and is sent to the user at the first next send from the module.

[EXCHANGE-POSITIVE] `OK`

The property holds.

[NO DUST POSITIONS] `OK`

The property holds.

[INTERNAL-PRECISION] All calculations are done in `PrecDec`, and the conversion only happens when interfacing with external accounts.

1. `Deposit`: `OK`, `NOT OK`
   - The value of issued shares is truncated↗ into `math.Int` , reducing the amount that the user would be able to claim at a later point. Since `SharesIssued/SharesToWithdraw` are mentioned in the interface, it is understandable that they themselves remain of type `Int`. The user loss exists here and it scales with number of deposits.
   - Other than that, the interface is correct: all internal calculations are in `PrecDec`, which only gets converted to `Int` when transferring from user to the module↗
2. `Withdraw`: `OK`
   - Shares (`Int`) are turned into `PrecDec` reserves in the `RedeemValue` function↗.
   - The same comment as for `Deposit`: shares amounts could also be augmented with fractional parts
3. `PlaceLimitOrder`: `OK`, `NOT OK`
   - Placing a limit order still takes an `Int`↗. This is not forced, since what we want to place is `amountLeft`↗, which is a `PrecDec` as a result of the internal calculation.
4. `WithdrawFilledLimitOrder`: `OK`
5. `CancelLimitOrder`: `OK`
6. `MultiHopSwap`: `OK`

[BEHAVIOR-PRESERVATION] If a user was able for a transaction `tx` get value `x` before introducing fractional accounting, not (after introducing it) the user gets >=`x`.

1. Liquidity iteration stops prematurely or iterates unnecessarily `OK`

The threat that was inspected hear was if very small maker amounts would remained uncleaned, potentially causing either stopping too early (before exhausting all the taker amount) or iterating over those negligible amounts.

In some sense, there is a premature stopping, when the remaining taker denom cannot get us more than a single micro coin, though this is acknowledged and by design↗. We verified that it does not hurt, but it is indeed unnecessary.

Following are the reasons why the threat does not materialize:

- the outer loop of liquidity iteration breaks either when there are no more liquidity sources↗ or when the `limitPrice` is crossed↗. In either case, unrelated to maker reserves.
- the individual swap will clear the whole amount in the reserves↗, as long as the taker amount suffices. When it does not suffice, a small amount may remain, but the next swap will clear it.

[IMPLEMENTATION]

1. Panic in `safeAdd`↗ stemming from `SendFractionalCoinsFromModuleToAccount` or `SendFractionalCoins-FromAccountToModule` will not happen, since all coins are either a single-element array, or created through `NewPrecDecCoins`. Since they enter the storage sorted, they remain sorted after doing the `balance.Add(tokens...)` because of the sort at return↗. `OK`
2. We analyzed for unused code remaining in the codebase after changes. A couple of instances were found and reported in the `Miscellaneous code improvements` finding. `OK`
3. Migration implementation `OK`, `NOT OK`

   We analyzed the migration implementation and found the following

   1. store was initialized correctly, in particular `dec_` values.
   2. a migration was not registered, reported in the `Migrate5to6 not registered` finding.

4. Validation `OK`

   The new coin (`PrecDecCoin`) contains the necessary validation

   - `PrecDecCoin` validation↗ does denom validation, nil and negative amount checks.
   - `PredDecCoins` validation↗ is also present.
   - Same or similar validations are also done for `sdk.Coin` (code ref↗).

5. Users funds strictly separated `OK`, `NOT OK`

   - Each fractional token balance is stored under a key↗ that consists of user address and token denom.
   - The value stored represents the actual fractional balanace for that specific user and token combination.
   - When sending withdraw and other proto messages, the creator of the message must be the signer (code ref↗) and is subsequently used as the callerAddr from which the balance is deducted (code ref↗).
   - This design ensures there is no possibility that other users can initiate withdrawals, cancellations, or any other unwilling actions on behalf of other users.
   - The authentication mechanism prevents unauthorized access to other users' balances by requiring the message creator to be the signer.

6. grpc queries are updated `OK`
7. Tests are updated `OK`

   - TODO for adding precdec fields left here↗.

Findings

Finding	Type	Severity	Status
Migrate5to6 not registered	Implementation	Medium	Resolved
Inefficient fractional balance loading	Protocol	Medium	Resolved
PlaceMakerLimitOrder should take PrecDec	Implementation	Low	Acknowledged
Not accounting for fractional debt when sending from accounts to the module	Protocol	Low	Resolved
Pool shares are still truncated, disregarding the fractional part	Protocol	Low	Risk Accepted
Miscellaneous code improvements	Implementation	Informational	Resolved

Migrate5to6 not registered

Severity Medium **Impact** 3 - High **Exploitability** 1 - Low

Type Implementation **Status** Resolved

Description

Migration from v5 to v6 is not registered↗ in `module.go`. `Migrate5to6` is never called.

Problem scenarios

Upon migration, new store variables won't be initialized.

Recommendation

Register migration in `module.go:RegisterServices`.

Inefficient fractional balance loading

Severity Medium **Impact** 1 - Low **Exploitability** 3 - High

Type Protocol **Status** Resolved

Involved artifacts

- `x/dex/keeper/fractional_banker.go`↗

Description

The `SendFractionalCoinsFromModuleToAccount` function in the `FractionalBanker` loads all fractional balances for a user address on every transaction, regardless of which specific tokens are being processed. This creates an inefficient pattern where:

1. `GetFractionalBalance(ctx, address)` retrieves all fractional coins for the user.
2. New tokens are added to balance and whole new balance is processed through `RoundDownToWholeTokenAmounts`.
3. The complete fractional balance is stored back, even when only a subset of tokens changed.

The current implementation stores fractional balances as a single `FractionalBalance` struct containing a slice of all `PrecDecCoin` for a user, requiring full loading and saving on every operation.

Problem scenarios

Contracts acting on behalf of multiple users (like Mars integrations) will trigger this inefficient loading pattern frequently for potentially large number of users. A malicious user may create a large number of token pairs permissionlessly, thereby either

1. Harming other users of the same contract (who would have to pay for that gas), or
2. Blocking the contract from performing any operations (by exceeding the total gas allowance for a transaction).

Recommendation

Store fractional balances per token-user pair instead of per user:

1. Use `KVStore` with keys `user_address + token_denom` and values as individual `PrecDec` amounts.
2. Only load the specific token balance of user needed for each operation.
3. Update only the relevant token balance instead of the entire user balance.

PlaceMakerLimitOrder should take PrecDec

Severity Low **Impact** 1 - Low **Exploitability** 1 - Low

Type Implementation **Status** Acknowledged

Description

The function `PlaceMakerLimitOrder` still takes an `Int`↗. This is not necessary, since what we want to place is `amountLeft`↗, which is a `PrecDec` as a result of the internal calculation (which subsequently gets truncated to `Int`).

Problem scenarios

Loss of precision.

Recommendation

Change the interface of the `PlaceMakerLimitOrder` function.

Not accounting for fractional debt when sending from accounts to the module

Severity Low **Impact** 1 - Low **Exploitability** 1 - Low

Type Protocol **Status** Resolved

Description

There is an asymmetry when between two functions of the `FractionalBalance` struct:

- `SendFractionalCoinsFromModuleToAccount` takes into account any existing fractional debt that needs to be sent to the user
- `SendFractionalCoinsFromAccountToModule` does not take into account existing debt.

This results in fractional balances potentially accumulating to values greater than 1.

Problem scenarios

Loss of precision.

Recommendation

Account for the debt in both functions.

Pool shares are still truncated, disregarding the fractional part

Severity Low **Impact** 1 - Low **Exploitability** 1 - Low

Type Protocol **Status** Risk Accepted

Description

The value of shared issued after depositing is truncated↗ into `math.Int`, reducing the amount that the user would be able to claim at a later point. Since `SharesIssued/SharesToWithdraw` are mentioned in the interface, it is understandable that they themselves remain of type `Int`. However, the loss of precision could be handled similarly to other coins in this chain (by accounting for fractional imprecision).

Problem scenarios

The user loss which scales with number of deposits. The particular situation occurs only where a deposit happens on a pool key which already exists, thus reducing the likelihood of it occurring.

Recommendation

Add the similar mechanism of storing fractional debt (in shares) and returning it to the user at a later point.

Miscellaneous code improvements

Severity Informational Impact: Exploitability:

Type Implementation **Status** Resolved

In this finding, we list a number of small improvements to the code that do not present a security threat.

1. Misleading panic message↗ inside `BinarySearchPriceToTick` : the message should be "Can only lookup prices >= 1"
2. Wrong comment here↗: it should replace references to `reserves1` with references to `reserves0`. (The code is correct.)
3. The style creates some confusion between `amountLeft` and `amountInDec` here↗. It should either be that `amountInDec` is used until the subtraction↗ of `swapInCoin.Amount` happens (at which point the new variable `amountLeft` can be used, or we should just use the same name constantly.
4. `SwapOnDepositSlopToleranceBps`↗ is never used but it remains in the codebase.
5. This comment↗ doesn't align with the actual test scenario.

Status

Resolved (PR#947↗, PR#938↗).

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.

ImpactScore	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)

ImpactScore	Examples
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)

ExploitabilityScore	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.

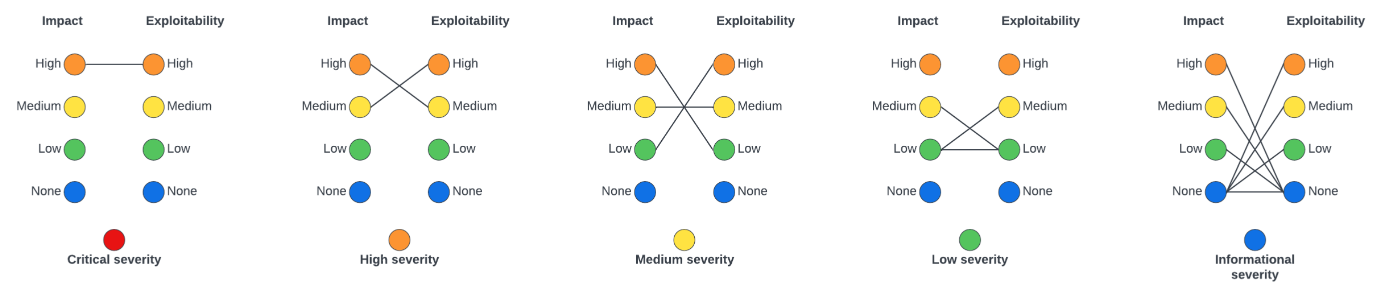

Figure 1: Severity classification

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.

SeverityScore	Examples
Critical	Halting of chain via a submission of a specially crafted transaction
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

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.