Fluence Deal Security Analysis

by Pessimistic

This report is public

April 2, 2024

Abstract

In this report, we consider the security of smart contracts of Fluence Deal project. Our task is to find and describe security issues in the smart contracts of the platform.

Disclaimer

The audit does not give any warranties on the security of the code. A single audit cannot be considered enough. We always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contracts. Besides, a security audit is not investment advice.

Summary

In this report, we considered the security of Fluence Deal smart contracts. We described the audit process in the section below.

The initial audit showed multiple issues of critical, medium and low severity.

After the audit, we reviewed multiple code versions. In those versions, the developers fixed all critical severity issues and most issues of medium and low severity. We discovered several new issues, which were subsequently fixed.

During the review, we identified the M03 issue as a false positive and removed it from the report.

The overall code quality is good. However, we strongly recommend implementing additional tests.

General recommendations

We recommend implementing more tests.

Project overview

Project description

For the audit, we were provided with Fluence Deal project on a public GitHub repository, commit fc00df476d6647259f4e882641d95645e62ab383. During the process of the review, we switched to a newer version of the code, commit 2503342115bdcd636f03cf44c6be21bebbee0656.

The scope of the audit included the whole repository.

The documentation for the project included the following link and private notion documents.

All 13 tests pass successfully. The code coverage could not be measured because of the size of the codebase.

The total LOC of audited sources is 2870.

Codebase update

After the initial audit, we reviewed the fixes across several commits. The most recent commit we reviewed is 2c7158231a39a2eabbc860d3116204f0af8cc2a8. For new issues, we have noted the commit hashes where they were found.

All 55 tests pass successfully. The code coverage could not be measured because of the size of the codebase.

Audit process

We started the audit on January 15, 2024, and finished on March 4, 2024.

We inspected the materials provided for the audit. Then, we contacted the developers for an introduction to the project. After a discussion, we performed preliminary research and identified those parts of the code and logic that required additional attention during the audit:

- Whether the logic of the codebase aligns with the documentation and developers' commentaries;
- Whether it is possible for providers to attack the deal or steal funds from the deal's owner;
- Whether the flow of the capacity commitment is valid and cannot be controlled by an attacker;
- Whether the logic of rewards distribution and fails count during capacity commitment is valid;
- Whether the code meets best practices;
- And many others.

For the audit, we split the codebase into several logical parts: **deal**, **utils**, **capacity**, and **market**. We started our review with the **deal** and **utils** parts and then moved to the **market** and **capacity** modules.

During the work, we stayed in touch with the developers, discussing confusing parts of the code. We contacted the developers multiple times to clarify suspicious parts and to share the existing results of the review up to that moment.

We manually analyzed all the contracts within the scope of the audit and checked their logic.

We scanned the project with the following tools:

- Static analyzer Slither;
- Our plugin Slitherin with an extended set of rules;
- Semgrep rules for smart contracts. We also shared the results with the developers in a text file.

We ran tests and tried to calculate the code coverage. The coverage calculation failed due to stack too deep error.

We compiled all the verified issues we found during the manual audit or discovered by automated tools into a private report.

After the initial audit, we discussed the results with the developers. It was decided to conduct a continuous audit, focusing on staying current with the latest code versions rather than reviewing a specific commit. Consequently, we reviewed the following commits:

- We began the recheck process with the commit 97e4786e7f48dc4d62595917316623fd85fa68a8. This commit did not contain fixes for the capacity module, so we focused our attention on the deal module;

- After that, we were provided with the commit 75cda77604c54b3cfb60f316ef17b583d18f3a5a. That update contained a non-final version of the refactored capacity module, in addition to small updates to the deal and util modules;
- For the final version, we were provided with commit number 2c7158231a39a2eabbc860d3116204f0af8cc2a8.

We could not calculate test coverage and run the Slitherin tool.

We reviewed the updated codebase, updated statuses for the issues, added comments, and wrote out new ones found during the manual review: M15, M16, M17, L22, L23, and N03. In addition to this, we identified the M03 issue as a false positive and removed it from the report.

Manual analysis

The contracts were completely manually analyzed, their logic was checked. Besides, the results of the automated analysis were manually verified. All the confirmed issues are described below.

Critical issues

Critical issues seriously endanger project security. They can lead to loss of funds or other catastrophic consequences. The contracts should not be deployed before these issues are fixed.

C01. Commitment snapshot can be called multiple times (fixed)

In the **Capacity.sol**, the `submitProof` function calls the `_commitCommitmentSnapshot` function multiple times, which leads to a situation where a user cannot submit more than the minimum required number of proofs. Once the user reaches the minimum required number of proofs, `cc.info.currentCUSuccessCount` increases by one. Consequently, the subsequent call to `_commitCommitmentSnapshot` will fail due to line 751: `reqSuccessCount - cc.info.currentCUSuccessCount`. `reqSuccessCount` is equal to `epoch - snapshotEpoch` (line 749), which equals zero since the snapshot was already committed in the current epoch. As a result, the transaction will fail due to an underflow error.

The issue has been fixed and is not present in the latest version of the code.

C02. Controversial checks occur when returning from a deal (fixed)

In **Offer.sol**, the `returnComputeUnitFromDeal` function is protected by the `onlyCapacity` modifier, which restricts calls exclusively to the capacity module. However, at lines 310-313, there are checks in place that permit only the `owner` of the deal or the `offer.provider` to make calls. These conditions are mutually exclusive, leading to an inevitable revert in one of the scenarios. As a result, compute units cannot be returned from deals.

The issue has been fixed and is not present in the latest version of the code.

C03. Controversial checks during matching (fixed)

In **Matcher.sol**, at line 117, there is a check to determine if the status of `commitmentId` is active. If `true`, the peer is skipped. However, later in the code, there is a call to `core.capacity().onUnitMovedToDeal(computePeer.commitmentId, unitId);` at line 459 in the **Offer** contract. In `Capacity.onUnitMovedToDeal`, a subsequent check after a snapshot commit verifies that the status is `CCStatus.Active` at line 203. Consequently, every unit with an active status is skipped, yet only an active unit can be moved to a deal. Furthermore, the check for active capacity is only performed if the `providersAccessType == IConfig.AccessType.WHITELIST` condition is met. This implies that it is possible to bypass this check for other types of whitelisting, potentially leading to unexpected behavior for units not participating in the **Capacity** module later in `_mvComputeUnitToDeal`.

The issue has been fixed and is not present in the latest version of the code.

C04. Deal cannot be started (fixed)

In **Deal.sol**, the `setWorker` function requires an `ACTIVE` status. However, the `getStatus` function returns an `INACTIVE` status in cases where the number of workers is below the required minimum.

The issue has been fixed and is not present in the latest version of the code.

C05. Unrestricted deal matching (fixed)

In **Matcher.sol**, the access to the `matchDeal` function is unrestricted, allowing anyone to match or re-match any deal, regardless of the deal creator's intentions.

The issue has been fixed and is not present in the latest version of the code.

C06. Incorrect boundary checks (fixed)

In **Capacity.sol**, the `_commitUnitSnapshot` function calculates the number of slashed epochs at line 691. If a user submits proofs for two consecutive epochs, the count will be `prevEpoch - lastMinProofsEpoch = prevEpoch - (prevEpoch - 1) = 1`, according to the calculation of `lastMinProofsEpoch` at line 707 in the `_commitUnitSnapshot` function. That means the user will be slashed for at least one epoch.

The issue has been fixed and is not present in the latest version of the code.

C07. Incorrect failed epoch calculation (fixed)

In **Capacity.sol**, in the `_failedEpoch` function, if a commitment fails at any point, the `failedEpoch` is set to `lastSnapshotEpoch_ + 1`. However, this approach may not accurately reflect the user's history of epochs without proofs prior to the failure. For example, consider a user submitting proof at epoch `0` and then committing a snapshot at epoch number `100`. Should the user fail, the `failedEpoch` would incorrectly be set to `1`, despite the actual failed epoch potentially being between `0` and `100`, as determined by `maxFailedRatio_`.

The issue has been fixed and is not present in the latest version of the code.

C08. Unrestricted access to onUnitMovedToDeal function (fixed)

In **Capacity.sol**, the `onUnitMovedToDeal` function lacks access restrictions, allowing anyone to call it. This could result in a scenario where a `unit` is falsely marked as `Inactive` without actually being in the `deal`.

The issue has been fixed and is not present in the latest version of the code.

C09. Unrestricted access to OnUnitReturnedFromDeal function (fixed)

In **Capacity.sol** `onUnitReturnedFromDeal` function lacks access restrictions, allowing anyone to call it.

The issue has been fixed and is not present in the latest version of the code.

C10. PeerId can be zero (fixed)

In the **Offer** contract, the `registerMarketOffer` function allows using `0` as a valid identifier for a peer. However, many parts of the codebase rely on the assumption that a `0` value indicates an object has not been created. We recommend ensuring that a `0` id is not considered a valid id for created structs.

The issue has been fixed and is not present in the latest version of the code.

C11. Risk of overwriting capacity information (fixed)

In **Capacity.sol**, the `depositCollateral` function does not verify whether collateral has already been deposited for a commitment. Consequently, anyone can overwrite existing information by calling `depositCollateral` again when the `delegator` is not specified. Moreover, the `createCommitment` function does not update the `peer.commitmentId` in the market.

The issue has been fixed and is not present in the latest version of the code.

C12. Public withdraw (fixed)

In the current version of the code, the `withdrawRewards` function in the **Deal** contract lacks access restriction. Consequently, it allows anyone to call it with any `computeUnitId`, potentially enabling them to claim another user's rewards.

The issue has been fixed and is not present in the latest version of the code.

C13. Unrestricted access to removeCUfromCC function (fixed)

In **Capacity.sol**, the `removeCUfromCC` function can be called by anyone for any compute unit.

The issue has been fixed and is not present in the latest version of the code. However, there is one scenario where delegator cannot retrieve their funds from the commitment. For the details, see N03 note.

C14. Potential inability to exit commitment (fixed)

In **Capacity.sol**, when a user attempts to finalize a commitment, there may be an unexpected revert, potentially trapping the user in that commitment. Lines 357-359 slash the user for failures. Line 357 calculates `totalCollateral` as `collateralPerUnit_ * unitCount`. Conversely, the slashed collateral is determined as `cc.info.totalCUFailCount * collateralPerUnit_`. Rewriting line 359 yields: `totalCollateral - slashedCollateral` equals to `collateralPerUnit_ * (unitCount - cc.info.totalCUFailCount)`. This implies that even if a unit fails for only two epochs, which could be less than `maxFailRatio`, the user might still be trapped in the commitment.

The issue has been fixed and is not present in the latest version of the code.

Medium severity issues

Medium severity issues can influence project operation in the current implementation. Bugs, loss of potential income, and other non-critical failures fall into this category, as well as potential problems related to incorrect system management. We highly recommend addressing them.

M01. Additional active unit count can be increased earlier than expected (fixed)

In **Capacity.sol**, the function `_commitCommitmentSnapshot` can be invoked multiple times for the same epoch. Consequently, the `cc.info.nextAdditionalActiveUnitCount` value might be added to the current active unit count within the same epoch if the user returns from a deal and `_commitCommitmentSnapshot` is called again.

The issue has been fixed and is not present in the latest version of the code.

M02. Commit during expired epoch (fixed)

In **Capacity.sol**, line 743 contains the expression: `if (epoch > expiredEpoch)`. However, the `expiredEpoch` should be considered as expired.

The issue has been fixed and is not present in the latest version of the code.

M04. Documentation

In the project, several concepts are not clearly articulated in the documentation:

- According to the documentation, "If balance gets lower than `minBalance`, an event for providers should be emitted, and the deal should become `INACTIVE`". However, this is not observed in the current implementation;
- According to the developers, a user can create only one worker per unit. Yet, the code does not enforce this limitation. Due to this, we recommend verifying the number of workers against the `targetWorkers` value in both the `addComputeUnit` and `setWorker` functions;
- In the current implementation, a provider can exit at the start of an epoch and still receive rewards;
- The **Multicall3** contract includes non-view calls, which contradicts comments indicating it is "used only for batch reading from Fluence frontends." Furthermore, we recommend paying close attention to the security considerations associated with the **Multicall3** contract.

M05. Early ending of the deal (fixed)

In **Deal.sol**, consider a scenario where a deal is operating with fewer workers than `targetWorkers`. If, in the last epoch, there are sufficient funds for the current number of workers but not enough for `targetWorkers`, then adding a new worker could immediately render the deal inactive (assuming the status operation is correct, see M14). This scenario could occur even though the funds would have been sufficient for a smaller number of workers.

The issue has been fixed at commit b42493c7a628445c0d088a04bd3bab498fae88b0 and is not present in the latest version of the code.

M06. Division by zero (fixed)

In **Capacity.sol**, line 712 can revert due to division by zero. This occurs because the `totalSuccessProofs` value, updated after a unit is committed, equals zero during the first commitment of a unit.

Furthermore, the operation at line 712 involves multiplication after division, resulting in a loss of precision.

The issue has been fixed and is not present in the latest version of the code.

M07. Peer ids are passed as arguments (addressed)

In **Offer.sol**, the `registerMarketOffer` function requires callers to specify peers themselves. This approach introduces a vulnerability where an individual with already registered peers can front-run new peers, utilizing their IDs as arguments for offer registration. Consequently, this action will cause the initial transaction to revert since the proposed IDs will have already been taken. This scenario could advantage existing peers by reducing competition for deals and capacity rewards.

According to the developers, a frontrun attack is not possible on the chain in use.

M08. **Project Roles** (addressed)

The owners of the project have the following powers:

- Changing the implementations of the contracts;
- They are responsible for setting FLT price;
- They are able to set commitment difficulty and various constants: `usdCollateralPerUnit`, `usdTargetRevenuePerEpoch`, `minDuration`, `minRewardPerEpoch`, `maxRewardPerEpoch`, `vestingPeriodDuration`, `vestingPeriodCount`, `slashingRate`, `minRequiredProofsPerEpoch`, `maxProofsPerEpoch`, `withdrawEpochesAfterFailed`, `maxFailedRatio`, `minDealDepositedEpoches`, `minDealRematchingEpoches`;
- The owners are responsible for whitelisting providers.

The owners of the deal have the capability to:

- Deposit and withdraw funds;
- Stop the deal;
- Whitelist users for the deal.

In addition, the **Matcher** contract is responsible for matching providers to deal owners. While this functionality is currently unrestricted, we recommend limiting access to it (refer to issue C05).

In the current implementation, the system depends heavily on the owners of the project. Thus, there are scenarios that can lead to undesirable consequences for the project and its users, e.g., if admin's private keys become compromised.

> *Comment from the developers:* *The owner of the contracts will be DAO.*

M09. **Difficulty changes immediately** (fixed)

In the **CapacityConst** contract, the `setDifficulty` function updates the difficulty. According to line 217, `difficultyChangeEpoch` is set to `core.currentEpoch() + 1`. However, the difficulty changes immediately because of how the `difficulty` function is implemented. Per line 163, if `constantsStorage.difficultyChangeEpoch >= core.currentEpoch()`, then `nextDifficulty` is returned.

The issue has been fixed and is not present in the latest version of the code.

M10. **Payment for the inactive epoch** (fixed)

In the **Deal** contract, if a user removes a computing unit causing the number of workers to fall below the required minimum, the deal becomes inactive. However, in the `_preCommitPeriod` function, payment will still be made.

The issue has been fixed and is not present in the latest version of the code.

M11. Unrestricted access (fixed)

In **Capacity.sol**, there is no restriction on who can delete a commitment immediately after its creation. This logic could facilitate a griefing attack, resulting in commitment rewards being distributed among a reduced number of users.

The issue has been fixed and is not present in the latest version of the code.

M12. User loses all rewards on resetting worker (fixed)

In **Deal.sol**, when a user calls the `setWorker` function for a second time, any previously accumulated rewards are lost, according to lines 349–350.

The issue has been fixed and is not present in the latest version of the code.

M13. User continues receiving rewards after removing computing unit (fixed)

In **Deal.sol**, the amount of rewards due for work is stored in the variable `dealStorage.cUnitPaymentInfo[computeUnitId]`. However, only the `getRewardAmount`, `withdrawRewards`, and `setWorker` functions interact with this variable. Consequently, when a compute unit is removed, the user's reward information remains unchanged, permitting the user to continue collecting rewards.

The issue has been fixed and is not present in the latest version of the code.

M14. Wrong status calculation (fixed)

In **Deal.sol**, the `getStatus` function incorrectly calculates the boundaries of the active epoch at line 161. The `_calculateMaxPaidEpoch` function indicates that `maxPaidEpoch` is not considered within the deal's active boundaries. Nonetheless, the `getStatus` function returns an `ACTIVE` status when `currentEpoch` is equal to `maxPaidEpoch`, as seen at line 153 in the `getStatus` function.

The issue has been fixed and is not present in the latest version of the code.

M15. Possible rewards distribution after the end of the deal (fixed)

The value of `maxPaidEpoch` variable of the **Deal** can be extended even after the call of the `stop` function by the owner. `Deal._postCommitPeriod` function is called even after the end of the deal, e.g., when the `withdarRewards` function is called. Thus, the value of `maxPaidEpoch` will be changed, and the units will acquire new rewards.

The issue was found during the review of commit 97e4786e7f48dc4d62595917316623fd85fa68a8. It has been fixed later and is not present in the latest version of the code.

M16. Unit will not receive rewards if they rejoin the deal (fixed)

The value of `lastWorkedEpoch` for a specific unit is not discarded when the unit leaves the deal. If the unit decides to rejoin the deal, then they will not get new rewards as their `lastWorkedEpoch` will be less than `snapshotEpoch` on line 451 of the **Deal** contract.

The issue was found during the review of commit 75cda77604c54b3cfb60f316ef17b583d18f3a5a. It has been fixed later and is not present in the latest version of the code.

M17. Returned units are not accounted for during the fails calculation (fixed)

When the unit returns from the deal, `nextAdditionalActiveUnitCount` is incremented to add all the returned units on the following epoch. These units are added only when `Capacity._preCommitCommitmentSnapshot` is called in the future epoch. However, if this epoch is not the next epoch after the return, these additional units must be accounted for fails for all the epochs they did not work except the epoch when they were added.

The issue was found during the review of commit 75cda77604c54b3cfb60f316ef17b583d18f3a5a. It has been fixed later and is not present in the latest version of the code.

Low severity issues

Low severity issues do not directly affect project operation. However, they might lead to various problems in future versions of the code. We recommend fixing them or explaining why the team has chosen a particular option.

L01. Unused structure field (fixed)

In the **Offer** contract, the `approved` field in the `ProviderInfo` structure is not used and always stores `false` value. Consider removing it.

The issue has been fixed and is not present in the latest version of the code.

L02. Memory location optimization (fixed)

In the **Offer** contract, in the `_addComputeUnitsToPeer` function, the location of the `unitIds` argument can be optimized by changing it to `calldata`.

The issue has been fixed and is not present in the latest version of the code.

L03. Duplicated code (fixed)

In the **Deal.sol** contract, the `withdrawRewards` and `getRewardAmount` functions share duplicated code segments.

The issues have been fixed and are not present in the latest version of the code.

L04. Duplicated check (fixed)

In **Capacity.sol**, there is a check at line 278 that ensures the provider address is not zero. However, similar checks are also performed at lines 275 and 279.

The issue has been fixed and is not present in the latest version of the code.

L05. External call inside for-loop (fixed)

In the **Offer** contract, in the `getComputeUnits` function, the same external contract call is executed on each iteration of the for-loop. Consider making it once before the loop.

The issue has been fixed and is not present in the latest version of the code.

L06. Gas optimization (fixed)

In the **LinkedListWithUniqueKeys** contract, in the `_has` function, it is sufficient to compare the `key` with either `self._first` or `self._last`.

The file is not present in the latest version of the code.

L07. Possibly incorrect status after returning from a deal (fixed)

In the **Capacity** contract, when a compute unit is returned from a deal, the `startEpoch` information is updated only in the `market` module. It would seem logical for the `Capacity.getStatus` function to return the `CCStatus.WaitStart` status as well. However, this does not occur because the `startEpoch` in the `capacity` module remains unchanged.

The issue has been fixed and is not present in the latest version of the code.

L08. Missing check (fixed)

We recommend adding a check to ensure the `minWorkers` variable is greater than zero during deal initialization.

The issue has been fixed and is not present in the latest version of the code.

L09. No effect when assigning (fixed)

In **Capacity.sol**, the statements `peer.commitmentId = bytes32(0x00);` at lines 307 and 325 do not have the intended effect because the `peer` variable is stored in memory.

The issue has been fixed and is not present in the latest version of the code.

L10. No need to update length (fixed)

In **Offer.sol**, there is no need to update the `computePeer.unitCount` variable at line 415, as it is already updated with the `_addComputeUnitsToPeer` call.

The issue has been fixed and is not present in the latest version of the code.

L11. Not all inherited contracts are initialized (fixed)

In **GlobalConst.sol**, the **OwnableUpgradableDiamond** and **EpochController** contracts are inherited but are not initialized.

The issues have been fixed and are not present in the latest version of the code.

L12. Redundant operation (fixed)

In **BytesConverter.sol**, in the `toBytes32` function, the `& 0xFF` operation has no effect.

The issue has been fixed and is not present in the latest version of the code.

L13. Return values of the functions are not used (fixed)

The `remove` and `add` functions in the **EnumerableSet** library return indicators of operation success. We recommend implementing explicit checks for these return values.

The issues have been fixed and are not present in the latest version of the code.

L14. Revert due to underflow

In **Deal.sol**, lines 252 and 264 can revert due to underflow. We recommend implementing reverts with explicit reasons to improve the user experience.

L15. TODO commentary (fixed)

The **Matcher** contract contains sections marked as TODO. We recommend addressing these areas before proceeding with deployment.

The issue has been fixed and is not present in the latest version of the code.

L16. Unreachable code (commented)

In the **GlobalConst** contract, within the `setConstant` function, the `else` branch is unreachable due to the current implementation of the `ConstantType` enum.

The same issue is present in the **CapacityConst** contract.

According to the developers, this safety check is intentional for contract upgrades.

L17. Unused imports

The **Deal.sol** file includes several unused imports: **Initializable.sol** and **IConfig.sol**.

L18. Unused ComputeProvidersList library (fixed)

The **ComputeProvidersList** library is implemented but is not used anywhere in the project.

The file is not present in the latest version of the code.

L19. Unused LinkedListWithUniqueKeys library (fixed)

In **Matcher.sol**, the **LinkedListWithUniqueKeys.sol** file is imported but not used. Consider removing this unused import.

The file is not present in the latest version of the code.

L20. Unused storage variables (fixed)

In the project, several contracts contain `_storage` variables that are never used.

The issues have been fixed and are not present in the latest version of the code.

L21. Use safeApprove for unknown tokens (fixed)

We recommend using the `safeApprove` function when working with the `paymentToken` in **DealFactory.sol**.

The issue has been fixed and is not present in the latest version of the code.

L22. Cache maxProofsPerEpoch (fixed)

In the `Capacity.submitProof`, the value of `minProofsPerEpoch` is cached. However, the value of `maxProofsPerEpoch` can be changed during the epoch. Consider cache `maxProofsPerEpoch` to specify equal parameters for all the participants during the single epoch.

The issue has been fixed and is not present in the latest version of the code.

L23. Commitment cannot be finished immediately after the return from deal (fixed)

When the unit is returned from the deal, the snapshot is updated as `unitInfo.lastSnapshotEpoch = currentEpoch;` at line 585. However, the unit can be removed from the commitment only when `true` is returned at line 449 of `removeCUFromCC` function. When the unit is returned within the `returnCUFromCC` function, `false` will be returned from snapshot because of `snapshotEpoch <= lastSnapshotEpoch` check in `_commitUnitSnapshot` function.

The issue has been fixed and is not present in the latest version of the code.

Notes

N01. Deal stop code logic (fixed)

In the current implementation of the **Deal** contract, the `stop` function cannot be invoked when the deal status is `INACTIVE`. Consider introducing logic that allows for the withdrawal of funds when the number of workers is below the required minimum.

The issue has been fixed and is not present in the latest version of the code.

N02. Possible reward withdrawal blocking by delegator (fixed)

In **Capacity.sol**, when a delegator is not explicitly specified, anyone can contribute liquidity to a commitment. This situation opens the door for a potential griefing attack by a malicious delegator. They could select a responsible provider, resulting in both the provider and delegator receiving rewards. However, once the commitment ends, the delegator could cause the `withdrawRewards` function calls to fail, since the rewards for the delegator and provider are not separated. Consequently, the delegator might demand additional compensation from the provider, since the delegator did not lose any funds. To mitigate this risk, we suggest implementing a mechanism to separate the withdrawal processes for delegator and provider rewards.

The issue has been fixed and is not present in the latest version of the code.

N03. Delegator cannot withdraw collateral while there are units in deals (new)

The delegator can withdraw their collateral only when all the units are removed from the commitment using `Capacity.removeCUFromCC`. However, the function can be called only by the provider.

The developers introduced some changes, and now it is possible for anyone to remove units when they are not in deals. However, delegator cannot remove units from deals if the capacity has ended, so some actions from the provider are still required.

This analysis was performed by Pessimistic:

Pavel Kondratenkov, Senior Security Engineer
Oleg Bobrov, Security Engineer
Egor Dergunov, Junior Security Engineer
Konstantin Zherebtsov, Business Development Lead
Irina Vikhareva, Project Manager
Alexander Seleznev, Founder

April 2, 2024