Quantstamp

Luganodes Pectra Batch Contract

Executive Summary

This audit report was prepared by Quantstamp, the leader in blockchain security.

Type	Validator Management
Timeline	2025-04-28 through 2025-05-02
Language	Solidity
Methods	Architecture Review, Unit Testing, Functional Testing, Computer-Aided Verification, Manual Review
Specification	None
Source Code	• https://github.com/Luganodes/Pectra-Batch-Contract #f63befb
Auditors	• Rabib Islam Senior Auditing Engineer • Yamen Merhi Auditing Engineer • Tim Sigl Auditing Engineer

Documentation quality	High
Test quality	High
Total Findings	3 Fixed: 3
High severity findings	0
Medium severity findings	0
Low severity findings	3 Fixed: 3
Undetermined severity findings	0
Informational findings	0

Summary of Findings

The presently audited contract is designed for Ethereum validator management. In particular, it aids in batch consolidation, batch withdrawals, and batch switching from type 1 withdrawal credentials to type 2 withdrawal credentials. It is intended to be delegated to by an EOA via a capability introduced in EIP-7702 as part of the Pectra hard fork.

No severe vulnerabilities were found. Those low-severity vulnerabilities that were found have relatively simple fixes.

Regarding the test suite, we recommend making use of Pectra-related capabilities in Foundry.

Update: All the issues were addressed, and the test suite was improved to include e.g. delegation using EIP-7702.

ID	DESCRIPTION	SEVERITY	STATUS
LUGA-1	Batch Operations Do Not Use Current Request Fee Which May Lead to Overpayment	• Low	Fixed
LUGA-2	Lack of Withdrawal Amount Validation May Lead to Unexpected Outcomes	• Low	Fixed
LUGA-3	EOA-Behaviour Mismatch	• Low	Fixed

Assessment Breakdown

Quantstamp's objective was to evaluate the repository for security-related issues, code quality, and adherence to specification and best practices.

Possible issues we looked for included (but are not limited to):

- Transaction-ordering dependence
- Timestamp dependence
- Mishandled exceptions and call stack limits
- Unsafe external calls
- Integer overflow / underflow
- Number rounding errors
- Reentrancy and cross-function vulnerabilities
- Denial of service / logical oversights
- Access control
- Centralization of power
- Business logic contradicting the specification
- Code clones, functionality duplication
- Gas usage
- Arbitrary token minting

Methodology

1. Code review that includes the following
    1. Review of the specifications, sources, and instructions provided to Quantstamp to make sure we understand the size, scope, and functionality of the smart contract.
    2. Manual review of code, which is the process of reading source code line-by-line in an attempt to identify potential vulnerabilities.
    3. Comparison to specification, which is the process of checking whether the code does what the specifications, sources, and instructions provided to Quantstamp describe.
2. Testing and automated analysis that includes the following:
    1. Test coverage analysis, which is the process of determining whether the test cases are actually covering the code and how much code is exercised when we run those test cases.
    2. Symbolic execution, which is analyzing a program to determine what inputs cause each part of a program to execute.
3. Best practices review, which is a review of the smart contracts to improve efficiency, effectiveness, clarity, maintainability, security, and control based on the established industry and academic practices, recommendations, and research.
4. Specific, itemized, and actionable recommendations to help you take steps to secure your smart contracts.

Scope

Files Included

- `src/*`

Repo: `https://github.com/Luganodes/Pectra-Batch-Contract`

Extensions: `sol`

Operational Considerations

Possibility of Sending Invalid Requests

When submitting requests to system contracts, the contracts do not check whether the caller is capable of handling the funds associated with the validator public keys. Instead, these checks are performed by consensus layer nodes. Therefore, submitting invalid requests is possible and will result in loss of Ether sent to the contracts.

Possibility of Consolidating to Validator of Another

When consolidating funds from a source validator to a target validator, the consensus layer nodes check whether the requesting execution-layer address is the same as the address in the source validator's withdrawal credential. However, it performs no such check with regard to the target validator's withdrawal credential. Therefore, it is possible to irretrievably consolidate funds to a validator that is not owned by the requestor.

To Be Used Only on Ethereum L1

The `Pectra` contract is to be used on Ethereum L1. It interfaces with mechanisms specifically present in the Pectra hard fork of Ethereum, such as withdrawals and consolidations. However, forks of Ethereum are implementing or have implemented aspects of

Pectra in their own nodes. However, the extent to which these nodes are similar to or different from Ethereum's nodes and the consequent impact of these differences on the functioning of the `Pectra` contract are not in scope in this audit.

Key Actors And Their Capabilities

Via EIP-7702, the contract is intended to be delegated to by an EOA. All function calls would be accessed by delegate calls, and hence the `onlySelf()` modifier would allow only the delegating EOA to call the functions of this contract.

Findings

LUGA-1

Batch Operations Do Not Use Current Request Fee Which May Lead to Overpayment

• Low ⓘ Fixed

Update

Marked as "Fixed" by the client.
Addressed in: `3ef8aefb35d95badecf21b4e6b0391bc07edf5bd` and `63ff13b26f3170da798fea1964f7d124a301a64a`.

Description: The following is noted in both EIP-7002 and EIP-7251:

```
Calls to the system contract require a fee payment defined by the current contract state.
Overpaid fees are not returned to the caller. It is not generally possible to compute the
exact required fee amount ahead of time. When adding a [withdrawal/consolidation] request
from a contract, the contract can perform a read operation to check for the current fee and
then pay exactly the required amount.
```

Currently, there are no checks in place to determine whether excess fees are about to be sent to the system contracts. Therefore, using the functions in `Pectra`, it is possible to send fees in excess of the required amount with no possibility of recovery.

Exploit Scenario:

1. A frontend or integration calculates the current per-request fee (e.g., 8 wei) via `staticcall` and sends a batch of 32 requests with `msg.value = 256 wei`.
2. Between the fee estimation and transaction inclusion, one or more blocks pass, and the required fee decreases to 5 wei.
3. The contract proceeds with `amountPerTx = msg.value / batchSize = 8 wei`, which exceeds the required 5 wei fee per call.
4. Each system contract call receives excess payment, resulting in an overpayment of 96 wei in total — none of which is refunded.

Recommendation: Enhance all batch functions to:

1. Perform a `staticcall` to the respective target contract to retrieve the current required fee.
2. Revert if the supplied `msg.value` is less than `fee * batchSize`.
3. Use the actual `fee` retrieved from the target contract in the `.call{value: fee}` invocation rather than computing `amountPerTx` manually.

LUGA-2

Lack of Withdrawal Amount Validation May Lead to Unexpected Outcomes

• Low ⓘ Fixed

Update

Marked as "Fixed" by the client.
Addressed in: `44e6d94df2ac1f251591a0d0f64d265946687815` `b40613bb1c4df691d479c296ab7860411af3eea9`, and `3e3145c7f8f3210ffb35d4d4dbff04004be49299`.

Description: In `batchELExit()`, although the length of the amount of an exit is validated, the exact value is not validated. However, validating this value may be worthwhile.

Suppose the value of the amount is `0`. In the consensus layer node logic, this will trigger a full exit. Although this may be desired, if `0` is entered not knowing this, then the requestor may unexpectedly exit their entire balance from the validator.

Now consider that the maximum amount of Ether a validator can hold in Pectra is `2048`. However, it is possible to submit a withdrawal request where the amount is greater than `2048` Ether. In such a case, if the validator has compounding withdrawal credentials, then as much Ether as possible will be withdrawn while maintaining the validator's minimum activation balance and pending balance to withdraw.

In either case, accidental inputs may lead to unexpected outcomes.

Recommendation: In addition to validating the length of the withdrawal amount, validate the value of the withdrawal amount. Ensure that if it `0`, then a request will be sent only if the user toggles a boolean flag in the parameters to `batchELExit()`. Moreover, consider skipping a withdrawal if the value of the amount exceeds `2048` Ether.

LUGA-3 EOA-Behaviour Mismatch

• Low ⓘ Fixed

Update
Marked as "Fixed" by the client.
Addressed in: `50b5dbd9dacc1436b781dccc6f43ba482038fb9b` and `085a6985fcc86843df1084c55d474bbe6dc67b3e`.

File(s) affected: `Pectra.sol`

Description: The documentation states:

```
Transaction Completion: The transaction finishes execution. The EOA's code delegation to
Pectra.sol remains in place.
```

Once delegation persists, the externally-owned account effectively becomes a contract account, yet the contract lacks:

- `receive()` — plain ETH transfers (`transfer` / `send`) revert.
- `fallback()` — calls with arbitrary calldata revert instead of no-oping as on a normal EOA.
- `onERC721Received` / `onERC1155Received` — safe token transfers fail.

This breaks long-established EOA behaviour and may surprise users, dApps, or tooling that expect the address to accept dust ETH or safe-transferred NFTs.

Recommendation: If perpetual delegation is intended, consider adding the following functions:

```
receive() external payable {}
fallback() external payable {}
function onERC721Received(...) external pure returns (bytes4) {
    return this.onERC721Received.selector;
}
function onERC1155Received(...) external pure returns (bytes4) {
    return this.onERC1155Received.selector;
}

function onERC1155BatchReceived(...) external pure returns (bytes4) {
    return this.onERC1155BatchReceived.selector;
}
```

However, if perpetual delegation is not intended, then removing delegation to the contract, at least for some time, may restore regular EOA behavior for that time.

Auditor Suggestions

S1 Use Custom Errors

Fixed

Update

Marked as "Fixed" by the client.
Addressed in: `29e649791b3ccb88894421bf883183f8dec25f80`.

Description: Compared to using `require()` with a string, the use of custom errors saves gas. Currently, only one custom error is used. However, all string-bearing `require()` statements can be replaced with custom errors.

In addition, in order to improve readability, custom errors can be implemented using `require()`, which is a feature introduced in Solidity 0.8.26.

Recommendation: Replace all `revert` and `require()` statements with `require()` statements that use custom errors.

S2 Magic Constants

Fixed

Update
Marked as "Fixed" by the client.
Addressed in: `5268e94a0e73f540cdab34c17ee357580cf8737f`.

Description: At a number of occasions, numeric literals are used in function logic without much description. It would be beneficial to instead use constants, and for each constant, write code comments explaining their purpose.

Recommendation: Replace numeric literals with constants.

S3 Gas Optimization

Fixed

Update
Marked as "Fixed" by the client.
Addressed in: `cccd9cffcfba3c124d9d606ba6feba994a86e8b6`.

Description: When incrementing through `for` loops, it is more expensive to use `i++` than `++i`.

Recommendation: Use `++i` instead of `i++` when incrementing through `for` loops.

S4 Remove Unused OpenZeppelin Dependency

Fixed

Update
Marked as "Fixed" by the client.
Addressed in: `ce0050b9f5f4865219d2db452eafc50ae75c7be6`.

Description: The OpenZeppelin contracts are installed via Foundry but are not used in the code.

Recommendation: Remove the unused dependency from the project to maintain a clean and minimal repository.

S5 Use `constant` Instead of `immutable` for Static Addresses

Fixed

Update
Marked as "Fixed" by the client.
Addressed in: `86a800f802f68ab8723cf1fe9656b503acaab1ba`.

File(s) affected: `Pectra.sol`

Description: The addresses `consolidationTarget` and `exitTarget` are known at compile time and are not set in the constructor. Declaring them as `constant` instead of `immutable` would reduce gas costs.

Recommendation: Use the `constant` keyword for these addresses, and optionally update their naming to uppercase for clarity:

```
address public constant CONSOLIDATION_TARGET
address public constant EXIT_TARGET
```

S6 Use `calldata` for External Function Parameters Instead of `memory`

Fixed

Update
Marked as "Fixed" by the client.
Addressed in: `f609e116233822a9531944699ec4257cfa281eaa`.

File(s) affected: `Pectra.sol`

Description: All external functions in the contract use the `memory` data location for dynamic input parameters (e.g., `bytes memory`, `bytes[] memory`). However, since these parameters are passed from outside the contract, they are initially stored in `calldata`. Copying them into `memory` introduces unnecessary gas costs.
The `calldata` location is ideal for external inputs — it is non-modifiable, non-persistent, and significantly more gas-efficient for read-only access.

Recommendation: Use `calldata` instead of `memory` for dynamic input types in all `external` functions where the parameters are only read and not modified.

Example From Code:

```
function batchConsolidation(bytes[] memory sourcePubkeys, bytes memory targetPubkey) external
payable onlySelf {
```

Recommended Improvement:

```
function batchConsolidation(bytes[] calldata sourcePubkeys, bytes calldata targetPubkey) external
payable onlySelf {
```

S7 Use Foundry EIP-7702 Cheatcodes to Simulate Live Delegation Behavior

Fixed

Update
Marked as "Fixed" by the client.
Addressed in: `cdb5ba8b0087098f3ffe7995bf6a8f66c7c5a152`.

File(s) affected: `Pectra.t.sol`

Description: To accurately reflect how the contract behaves when used via EIP-7702 delegation, tests should incorporate Foundry's delegation-specific cheatcodes. The `signAndAttachDelegation` cheatcode enables simulation of EOA-to-contract delegation in a single step, mimicking on-chain behavior introduced in EIP-7702.

Example:

```
vm.signAndAttachDelegation(address(pectra), EOA_PRIVATE_KEY);
```

This designates the next call as originating from an EOA delegated to the `Pectra` contract.

Recommendation: Leverage `signAndAttachDelegation` in the test suite to ensure contract behavior aligns with real-world delegated execution scenarios.

S8 Event Emission Improvements

Fixed

Update
Marked as "Fixed" by the client.
Addressed in: `d6be7c152bd7967ca5ada30b511d7a3c1a48f301`.

File(s) affected: `Pectra.sol`

Description: Event emission can improved on a number of items.
1. Failure events embed long English sentences plus raw `bytes` data; repeated emissions inside loops consume considerable gas and bloat logs.
2. `ConsolidationFailed` does not include the `targetPubkey`, limiting the usefulness of the event for debugging and analytics.
3. All failure events carry a `sender` field, but `onlySelf` restricts calls to a single address, making that parameter redundant gas overhead.

Recommendation: Implement the following changes:
1. Replace verbose strings with concise numeric reason codes or define separate event signatures per failure type and decode them off-chain to cut gas costs.
2. Add the `targetPubkey` parameter to `ConsolidationFailed`, ensuring both source and destination keys are emitted for full context.
3. Drop the `sender` argument from failure events to further reduce event size.

Definitions

- **High severity** – High-severity issues usually put a large number of users' sensitive information at risk, or are reasonably likely to lead to catastrophic impact for client's reputation or serious financial implications for client and users.
- **Medium severity** – Medium-severity issues tend to put a subset of users' sensitive information at risk, would be detrimental for the client's reputation if exploited, or are reasonably likely to lead to moderate financial impact.
- **Low severity** – The risk is relatively small and could not be exploited on a recurring basis, or is a risk that the client has indicated is low impact in view of the client's business circumstances.
- **Informational** – The issue does not post an immediate risk, but is relevant to security best practices or Defence in Depth.
- **Undetermined** – The impact of the issue is uncertain.
- **Fixed** – Adjusted program implementation, requirements or constraints to eliminate the risk.
- **Mitigated** – Implemented actions to minimize the impact or likelihood of the risk.
- **Acknowledged** – The issue remains in the code but is a result of an intentional business or design decision. As such, it is supposed to be addressed outside the programmatic means, such as: 1) comments, documentation, README, FAQ; 2) business processes; 3) analyses showing that the issue shall have no negative consequences in practice (e.g., gas analysis, deployment settings).

Appendix

File Signatures

The following are the SHA-256 hashes of the reviewed files. A file with a different SHA-256 hash has been modified, intentionally or otherwise, after the security review. You are cautioned that a different SHA-256 hash could be (but is not necessarily) an indication of a changed condition or potential vulnerability that was not within the scope of the review.

Files

Repo: `https://github.com/Luganodes/Pectra-Batch-Contract`

- `af4...1f5 ./src/Pectra.sol`

Toolset

The notes below outline the setup and steps performed in the process of this audit.

Setup

Tool Setup:
- Slither v0.10.0

Steps taken to run the tools:
1. Install the Slither tool: `pip3 install slither-analyzer`

2. Run Slither from the project directory: `slither .`

Automated Analysis

Slither

Slither identifies the use of numeric literals.

Test Suite Results

Overall, the test suite is rather good. However, it does not use Pectra delegation. See S4 for details.

Update: The test suite was bolstered from 23 tests to 32 tests, including e.g. delegation and retrieving fees.

```
Ran 32 tests for test/Pectra.t.sol:PectraTest
[PASS] testBatchConsolidation_Delegation() (gas: 62216)
[PASS] testBatchConsolidation_EmptySources() (gas: 19058)
[PASS] testBatchConsolidation_FailedCall() (gas: 1056691755)
[PASS] testBatchConsolidation_InvalidSourcePubkeyLength() (gas: 33368)
[PASS] testBatchConsolidation_InvalidTargetLength() (gas: 21897)
[PASS] testBatchConsolidation_Success() (gas: 57160)
[PASS] testBatchConsolidation_TooManySources() (gas: 146357)
[PASS] testBatchConsolidation_Unauthorized() (gas: 20212)
[PASS] testBatchELExit_Delegation() (gas: 55181)
[PASS] testBatchELExit_EmptyData() (gas: 17164)
[PASS] testBatchELExit_ExceedsMaximumAmount() (gas: 33289)
[PASS] testBatchELExit_FailedCall() (gas: 1056691311)
[PASS] testBatchELExit_FullExitWithAmount() (gas: 31572)
[PASS] testBatchELExit_InvalidPublicKeyLength() (gas: 30794)
[PASS] testBatchELExit_SuccessWithValidAmount() (gas: 50292)
[PASS] testBatchELExit_SuccessWithZeroAmount() (gas: 51131)
[PASS] testBatchELExit_TooManyValidators() (gas: 556674)
[PASS] testBatchELExit_Unauthorized() (gas: 18989)
[PASS] testBatchELExit_ZeroAmountWithoutConfirmation() (gas: 32089)
[PASS] testBatchSwitch_Delegation() (gas: 50453)
[PASS] testBatchSwitch_EmptyValidators() (gas: 17185)
[PASS] testBatchSwitch_FailedCall() (gas: 1056690367)
[PASS] testBatchSwitch_InvalidValidatorPubkeyLength() (gas: 28710)
[PASS] testBatchSwitch_Success() (gas: 45446)
[PASS] testBatchSwitch_TooManyValidators() (gas: 418726)
[PASS] testBatchSwitch_Unauthorized() (gas: 18314)
[PASS] testDelegation_FailsDueToOnlySelf() (gas: 23674)
[PASS] testDelegation_InvalidParameters() (gas: 21321)
[PASS] testDelegation_WrongPrivateKey() (gas: 25205)
[PASS] testGetFee_ConsolidationTarget() (gas: 12593)
[PASS] testGetFee_ExitTarget() (gas: 12590)
[PASS] testGetFee_FailedCall() (gas: 1040433000)
Suite result: ok. 32 passed; 0 failed; 0 skipped; finished in 125.56ms (1.02s CPU time)

Ran 1 test suite in 148.65ms (125.56ms CPU time): 32 tests passed, 0 failed, 0 skipped (32 total
tests)
```

Code Coverage

Code coverage is very good, with branch coverage exceeding 90%.

Update: Branch coverage has dipped slightly, from 91.18% to 90%.

```
╭----------------+----------------+----------------+----------------+---------------╮
| File           | % Lines        | % Statements   | % Branches     | % Funcs       |
+==================================================================================+
| src/Pectra.sol | 88.16% (67/76) | 90.00% (81/90) | 90.00% (27/30) | 60.00% (6/10) |
|----------------+----------------+----------------+----------------+---------------|
| Total          | 88.16% (67/76) | 90.00% (81/90) | 90.00% (27/30) | 60.00% (6/10) |
╰----------------+----------------+----------------+----------------+---------------╯
```

Changelog

- 2025-05-02 - Initial report
- 2025-05-07 - Final report

About Quantstamp

Timeliness of content

Notice of confidentiality

Links to other websites

Disclaimer

Quantstamp