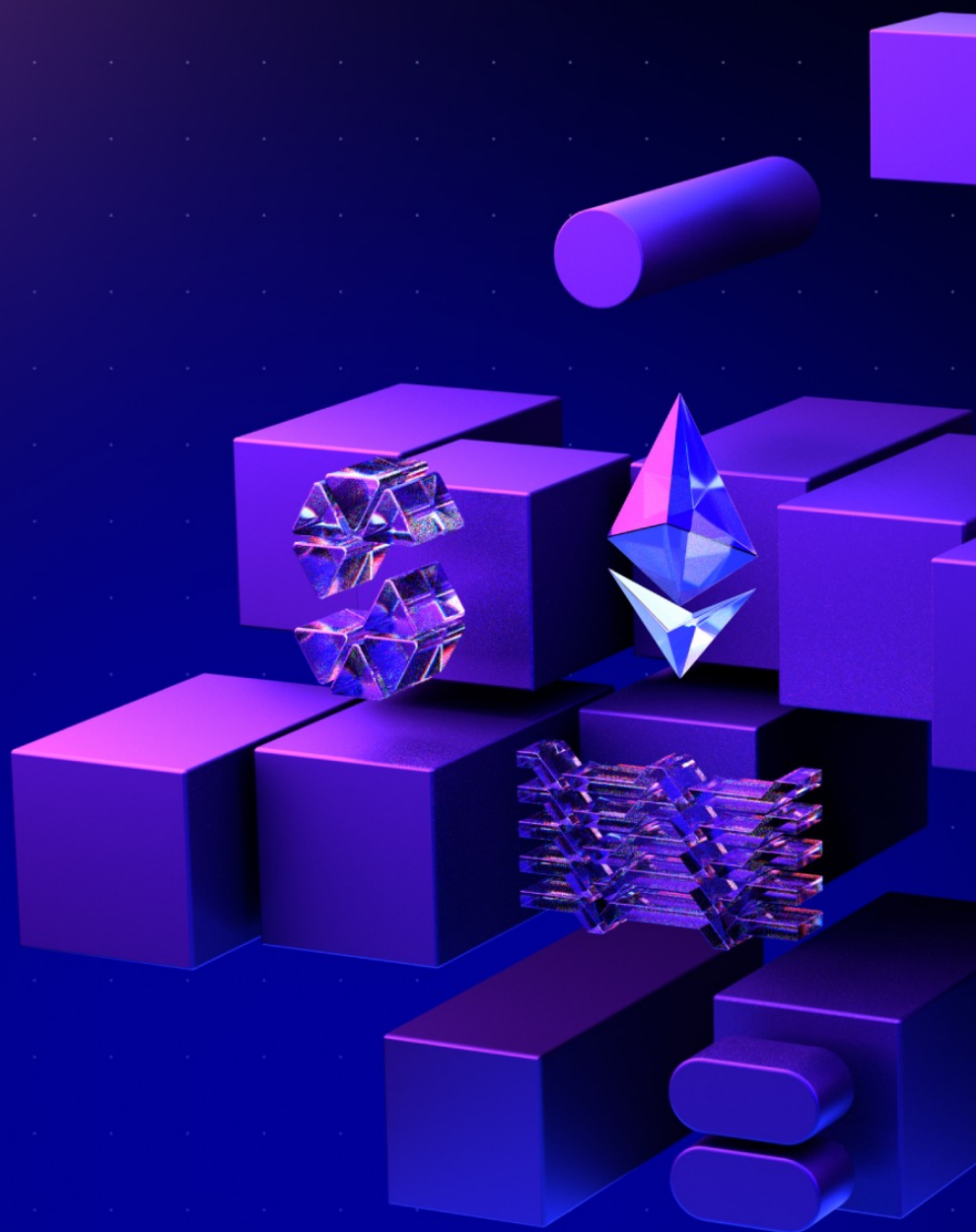
ackee
blockchain security

Everstake

ETH2 Batch Deposit Contract

21.11.2025



Audit Report

Contents

1. Document Revisions

1.0-draft	Draft Report	14.11.2025
1.0	Final Report	14.11.2025
1.1	Fix Review	21.11.2025

2. Overview

This document presents our findings in reviewed contracts.

2.1. Ackee Blockchain Security

Ackee Blockchain Security is an in-house team of security researchers performing security audits focusing on manual code reviews with extensive fuzz testing for Ethereum and Solana. Ackee is trusted by top-tier organizations in web3, securing protocols including Lido, Safe, and Axelar.

We develop open-source security and developer tooling Wake for Ethereum and Trident for Solana, supported by grants from Coinbase and the Solana Foundation. Wake and Trident help auditors in the manual review process to discover hardly recognizable edge-case vulnerabilities.

Our team teaches about blockchain security at the Czech Technical University in Prague, led by our co-founder and CEO, Josef Gattermayer, Ph.D. As the official educational partners of the Solana Foundation, we run the School of Solana and the Solana Auditors Bootcamp.

Ackee's mission is to build a stronger blockchain community by sharing our knowledge.

Ackee Blockchain a.s.
Rohanske nabrezi 717/4
186 00 Prague, Czech Republic
https://ackee.xyz
hello@ackee.xyz

2.2. Audit Methodology

1. **Verification of technical specification**
   The audit scope is confirmed with the client, and auditors are onboarded to the project. Provided documentation is reviewed and compared to the audited system.

2. **Tool-based analysis**
   A deep check with Solidity static analysis tool Wake in companion with Solidity (Wake) extension is performed, flagging potential vulnerabilities for further analysis early in the process.

3. **Manual code review**
   Auditors manually check the code line by line, identifying vulnerabilities and code quality issues. The main focus is on recognizing potential edge cases and project-specific risks.

4. **Local deployment and hacking**
   Contracts are deployed in a local Wake environment, where targeted attempts to exploit vulnerabilities are made. The contracts' resilience against various attack vectors is evaluated.

5. **Unit and fuzz testing**
   Unit tests are run to verify expected system behavior. Additional unit or fuzz tests may be written using Wake framework if any coverage gaps are identified. The goal is to verify the system's stability under real-world conditions and ensure robustness against both expected and unexpected inputs.

6. **Wake-AI assisted vulnerability discovery**
   As the last step, the scope is checked against Wake AI, an LLM-powered audit tool, to identify potentially missed vulnerabilities. This step is executed at the end of the audit process to avoid distracting auditors from manual review.

2.3. Finding Classification

A *Severity* rating of each finding is determined as a synthesis of two sub-ratings: *Impact* and *Likelihood*. It ranges from *Informational* to *Critical*.

If we have found a scenario in which an issue is exploitable, it will be assigned an impact rating of *High*, *Medium*, or *Low*, based on the direness of the consequences it has on the system. If we haven't found a way, or the issue is only exploitable given a change in *configuration* (system settings or parameters, such as deployment scripts, compiler configurations, using multi-signature wallets for owners, etc.) or given a change in the codebase, then it will be assigned an impact rating of *Warning* or *Info*.

Low to *High* impact issues also have a *Likelihood*, which measures the probability of exploitability during runtime.

The full definitions are as follows:

Severity

		Likelihood			
		High	**Medium**	**Low**	**N/A**
Impact	**High**	Critical	High	Medium	-
	Medium	High	Medium	Low	-
	Low	Medium	Low	Low	-
	Warning	-	-	-	Warning
	Info	-	-	-	Info

Table 1. Severity of findings

Impact

- **High** - Code that activates the issue will lead to undefined or catastrophic consequences for the system.
- **Medium** - Code that activates the issue will result in consequences of serious substance.
- **Low** - Code that activates the issue will have outcomes on the system that are either recoverable or don't jeopardize its regular functioning.
- **Warning** - The issue cannot be exploited given the current code and/or *configuration*, but could be a security vulnerability if these were to change slightly. If we haven't found a way to exploit the issue given the time constraints, it might be marked as a "Warning" or higher, based on our best estimate of whether it is currently exploitable.
- **Info** - The issue is on the borderline between code quality and security. Examples include insufficient logging for critical operations. Another example is that the issue would be security-related if code or *configuration* was to change.

Likelihood

- **High** - The issue is exploitable by virtually anyone under virtually any circumstance.
- **Medium** - Exploiting the issue currently requires non-trivial preconditions.
- **Low** - Exploiting the issue requires strict preconditions.

2.4. Review Team

The following table lists all contributors to this report. For authors of the specific revision, see the “Revision team” section in the respective “Report revision” chapter.

Member’s Name	Position
Lukáš Rajnoha	Lead Auditor
Josef Gattermayer, Ph.D.	Audit Supervisor

2.5. Disclaimer

We’ve put our best effort to find all vulnerabilities in the system, however our findings shouldn’t be considered as a complete list of all existing issues. The statements made in this document should not be interpreted as investment or legal advice, nor should its authors be held accountable for decisions made based on them.

3. Executive Summary

Everstake is a blockchain infrastructure provider operating validators across multiple networks. The ETH2 Batch Deposit Contract allows consolidating multiple validator deposits into a single transaction and forwarding them atomically to the official ETH2 Deposit Contract.

Revision 1.0

Everstake engaged Ackee Blockchain Security to perform a security review of Everstake ETH2 Batch Deposit Contract with a total time donation of 2 engineering days in a period between November 11 and November 14, 2025, with Lukáš Rajnoha as the lead auditor.

The audit was performed on the commit `c2c12ba`[1] in the contracts repository and the scope was the following:

- `contracts/ETH2BatchDepositConsolidation.sol`

The contract in scope was also deployed at the `0x4ff41fa0f4e77129c4c0607994050473c2067e6d` address on Mainnet.

We began our review using static analysis tools, including Wake. We then performed a thorough manual review of the code, especially focusing on integration with the canonical ETH2 Deposit Contract. During the review, we paid special attention to:

- ensuring no griefing or front-running attacks are possible;
- ensuring interactions with external contracts are correctly implemented;
- ensuring compatibility with recent Ethereum protocol updates;
- verifying the arithmetic of the system is correct;
- looking for common issues such as data validation.

At the end of the review, we engaged the Wake AI tool, which discovered the following issues: I2.

Our review resulted in 2 findings, all of Info severity and related to data validation.

Ackee Blockchain Security recommends Everstake to:

- ensure off-chain components are correctly implemented and properly verify payloads sent to the contract in scope;
- address all the reported issues.

See Report Revision 1.0 for the system overview and trust model.

Revision 1.1

Everstake engaged Ackee Blockchain Security to perform a fix review of the findings from the previous revision. Everstake provided a pull request PR #1 with the fixes.

The review was performed on November 21, 2025 on the commit `507a932`[2].

From the reported 2 findings, all were fixed. No new findings were reported.

[1] full commit hash: `c2c12bace005a912860129dd544abf1a02a39968`, link to commit

[2] full commit hash: `507a932059d71c4daeb53d91ecc26d2cb027609b`, link to commit

4. Findings Summary

The following section summarizes findings we identified during our review. Unless overridden for purposes of readability, each finding contains:

- *Description*
- *Exploit scenario* (if severity is low or higher)
- *Recommendation*
- *Fix* (if applicable).

Summary of findings:

Critical	High	Medium	Low	Warning	Info	Total
0	0	0	0	0	2	**2**

Table 2. Findings Count by Severity

Findings in detail:

Finding title	Severity	Reported	Status
I1: Limited deposit validation	Info	1.0	Fixed
I2: Missing cumulative deposit funds check	Info	1.0	Fixed

Table 3. Table of Findings

Report Revision 1.0

Revision Team

Member's Name	Position
Lukáš Rajnoha	Lead Auditor
Josef Gattermayer, Ph.D.	Audit Supervisor

System Overview

The `BatchDepositConsolidation` contract is a utility that consolidates multiple validator deposits into a single transaction by forwarding each entry to a configured ETH2 Deposit Contract. Callers provide a validity deadline, a single withdrawal credential applied to all entries, per-deposit amounts, and a batch of deposit data. The contract then validates basic input shape, and forwards each deposit atomically.

Trust Model

The contract is permissionless and introduces no additional trust assumptions beyond the official ETH2 Deposit Contract.

Findings

The following section presents the list of findings discovered in this revision. For the complete list of all findings, Go back to Findings Summary

I1: Limited deposit validation

Impact:	Info	Likelihood:	N/A
Target:	ETH2BatchDepositConsolidation.sol	Type:	Data validation

Description

The `BatchDepositConsolidation` contract is missing certain input parameter validation which could help prevent invalid deposits, namely:

- verifying withdrawal credentials (checking `withdrawAddress` is not a zero address, and ensuring `withdrawType` is a valid value);
- validating the number of deposits in the batch (ensuring it is not zero).

Listing 1. Excerpt from ETH2BatchDepositConsolidation

```
23 function batchDeposit(uint validUntil, address withdrawAddress, bytes1
   withdrawType, uint256[] calldata values, bytes calldata args) external
   payable {
24     require(
25         block.timestamp < validUntil,
26         "deposit data agreed upon deadline");
27     require(
28         args.length % depositArgsLength == 0,
29         "wrong input"
30     );
31     uint count = args.length / depositArgsLength;
32     require(count == values.length, "mismatched num of args");
33
34     uint signatureStart;
35     uint depositDataRootStart;
36     uint depositDataRootEnd;
37
38     bytes memory rawWithdrawAuthority = abi.encodePacked(withdrawType,
   hex"000000000000000000000000", withdrawAddress);
```

While some of the additional checks (e.g. validating the `withdrawType` input parameter) may risk reducing future compatibility if the Ethereum protocol

specification changes, other checks — such as ensuring `withdrawAddress` is not a zero address — are expected to remain valid for the foreseeable future. Since invalid deposits may lead to loss of funds, adding any additional checks provides additional safeguards against accidental misconfiguration. The zero deposits check would only prevent empty transactions and is therefore not strictly necessary.

Recommendation

Add a zero address check for the `withdrawAddress` input parameter.

Fix 1.1

The issue was fixed by adding a zero address check for the `withdrawAddress` input parameter.

Listing 2. Excerpt from ETH2BatchDepositConsolidation

```
23 function batchDeposit(uint validUntil, address withdrawAddress, bytes1
   withdrawType, uint256[] calldata values, bytes calldata args) external
   payable {
24     require(
25         block.timestamp < validUntil,
26         "deposit data agreed upon deadline");
27     require(
28         args.length % depositArgsLength == 0,
29         "wrong input"
30     );
31     require(
32         withdrawAddress != address(0),
33         "zero address"
34     );
```

Go back to Findings Summary

I2: Missing cumulative deposit funds check

Impact:	Info	Likelihood:	N/A
Target:	ETH2BatchDepositConsolidation.sol	Type:	Data validation

Description

The deposit function does not verify that `msg.value` equals the sum of all `values` array elements. The official ETH2 Deposit Contract only enforces loose bounds on `msg.value` (minimum 1 ether and maximum 2^64-1 gwei). As a result, several issues can occur:

1. **Excess funds**: More ETH can be sent to the function than will be forwarded to the ETH2 Deposit Contract, leaving excess funds in the contract.
2. **Transaction failure**: The transaction will fail without a proper error if insufficient funds were passed in.
3. **Exploitation of existing balance**: If the contract already holds some ETH, a user can underpay by subtracting the contract's existing balance from the required deposit amount. The transaction will still succeed, effectively allowing the user to use the contract's balance to fund their deposit.

Any excess ETH left in the contract becomes inaccessible because there is no withdrawal mechanism. The only way to "use" these stranded funds is for future callers to underpay and rely on the contract's residual balance to cover the difference.

Recommendation

Add validation to ensure that `msg.value` equals the sum of all deposit values in the `values` array, which will prevent the issues mentioned above.

Fix 1.1

The issue was fixed by adding validation to ensure that `msg.value` equals the sum of all deposit values in the `values` array.

Listing 3. Excerpt from ETH2BatchDepositConsolidation

```
41 uint256 totalValue;
42
43 bytes memory rawWithdrawAuthority = abi.encodePacked(withdrawType,
   hex"000000000000000000000000", withdrawAddress);
44 for (uint j = 0; j < count; j++) {
45     unchecked
46     {
47         signatureStart = j * depositArgsLength + pubkeyLength;
48         depositDataRootStart = signatureStart + signatureLength;
49         totalValue += values[j];
50     }
51
52     depositContract.deposit{value: values[j] }(
53         args[j * depositArgsLength : signatureStart],
54         rawWithdrawAuthority,
55         args[signatureStart : depositDataRootStart],
56         // bytes32 depositDataRoot
57         bytes32(args[depositDataRootStart : depositDataRootStart +
   depositDataRootLength])
58     );
59 }
60
61 require(totalValue == msg.value, "value");
```

Go back to Findings Summary

Appendix A: How to cite

Please cite this document as:

Ackee Blockchain Security, Audit Report | Everstake: ETH2 Batch Deposit Contract, 21.11.2025.

Appendix B: Wake AI Findings

This section lists vulnerabilities identified by Wake AI, an LLM-powered audit tool used for AI-assisted vulnerability discovery during the audit. Wake AI leverages large language models to understand code context and reason about complex contract behavior, complementing manual review.

B.1. Discovered Findings

The following table contains true-positive findings identified by Wake AI. These findings are included regardless of whether they were also discovered independently by auditors during manual review.

Finding title	Severity	Reported	Discoverer
I2: Missing cumulative deposit funds check	Info	1.0	Wake AI, Auditor

Table 4. Table of findings identified by Wake AI

ackee
blockchain security

Thank You

Ackee Blockchain a.s.

Rohanske nabrezi 717/4
186 00 Prague
Czech Republic

hello@ackee.xyz