Blaize.Security

March 10th, 2023 / V. 1.0

everstake

EVERSTAKE

SMART CONTRACT AUDIT

TABLE OF CONTENTS

AUDIT RATING

Everstake contract's source code was taken from the repository provided by the Everstake team.

SCORE **10** /10

🟥 🟥 🟥 🟥 🟥 🟥 🟥 🟥 🟥 🟥

The scope of the project is **Everstake** set of contracts during 1st audit iteration:

PoolB2B.sol
ValidatorList.sol

Repository:
https://github.com/everstake/ETH-Staking-B2B-SC

Branch: master

Intial commit (audited):

- 94de9df108ff62907cbd66cc2c0d5968b8de3980

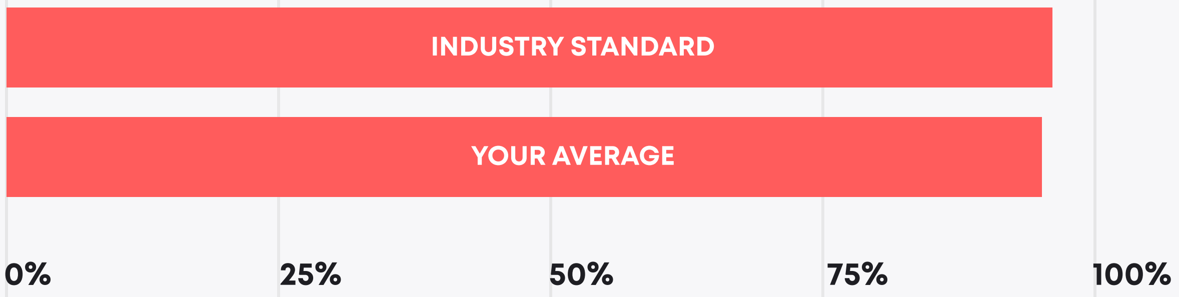
Final commit (post-audit):

- 20fbdaeb6b5310a218ea8a89c920dfb35feba815

TECHNICAL SUMMARY

In this report, we consider the security of the contracts for the **Everstake** protocol. Our task is to find and describe security issues in the smart contracts of the platform. This report presents the findings of the security audit of **Everstake** smart contracts conducted between **February 23rd, 2023 - March 10th, 2023**

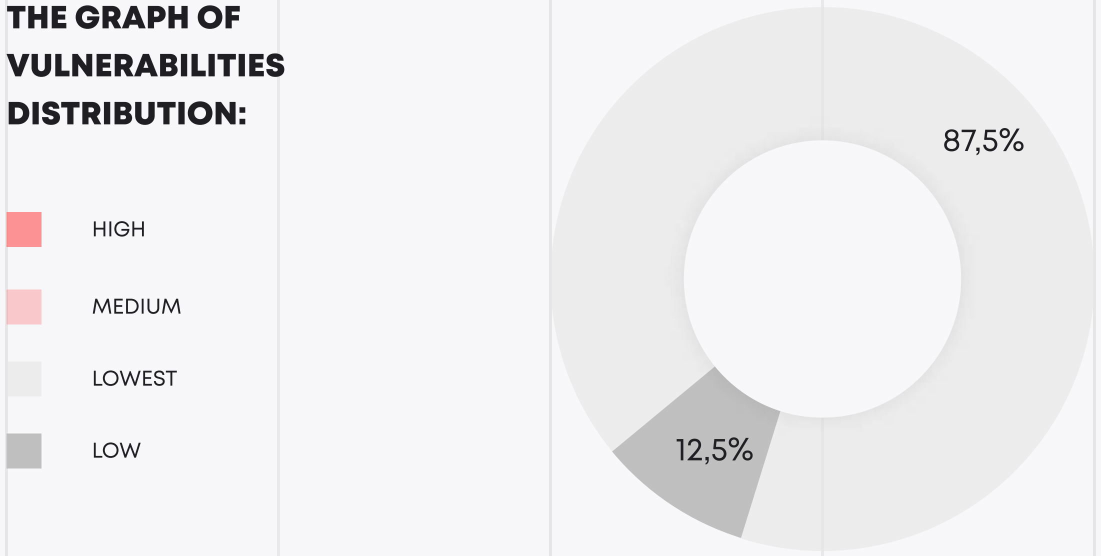
Testable code



Security team ensured, that the testable code corresponds the industry standard. That includes both coverage and manual tests for the uncovered logic.

The scope of the audit includes the unit test coverage, that bases on the smart contracts code, documentation and requirements presented by the Everstake team. Coverage is calculated based on the set of Hardhat framework tests and scripts from additional testing strategies. Though, in order to ensure a security of the contract Blaize.Security team recommends the Everstake team put in place a bug bounty program to encourage further and active analysis of the smart contracts.

THE GRAPH OF VULNERABILITIES DISTRIBUTION:



The table below shows the number of found issues and their severity. A total of 16 problems were found, most of which were connected to gas optimization. All of them were successfully fixed by the Everstake team.

	FOUND	FIXED/VERIFIED
Critical	0	0
High	0	0
Medium	0	0
Low	2	2
Lowest	14	14

SEVERITY DEFINITION

Critical

A system contains several issues ranked as very serious and dangerous for users and the secure work of the system. Needs immediate improvements and further checking.

High

A system contains a couple of serious issues, which lead to unreliable work of the system and migh cause a huge information or financial leak. Needs immediate improvements and further checking.

Medium

A system contains issues which may lead to mediumfinancial loss or users' private information leak. Needs immediate improvements and further checking.

Low

A system contains several risks ranked as relatively small with the low impact on the users' information and financial security. Needs improvements.

Lowest

A system does not contain any issue critical to the secure work of the system, yet is relevant for best

AUDITING STRATEGY AND TECHNIQUES APPLIED \ PROCEDURE

We have scanned this smart contract for commonly known and more specific vulnerabilities:

- Unsafe type inference;
- Timestamp Dependence;
- Reentrancy;
- Implicit visibility level;
- Gas Limit and Loops;
- Transaction-Ordering Dependence;
- Unchecked external call - Unchecked math;
- DoS with Block Gas Limit;
- DoS with (unexpected) Throw;
- Byte array vulnerabilities;
- Malicious libraries;
- Style guide violation;
- ERC20 API violation;
  Uninitialized state/storage/ local variables;
- Compile version not fixed.

Procedure

In our report we checked the contract with the following parameters:

- Whether the contract is secure;
- Whether the contract corresponds to the documentation;
- Whether the contract meets best practices in efficient use of gas, code readability;

Automated analysis:

Scanning contract by several public available automated analysis tools such as Mythril, Solhint, Slither and Smartdec. Manual verification of all the issues found with tools.

Manual audit:

Manual analysis of smart contracts for security vulnerabilities. Checking smart contract logic and comparing it with the one described in the documentation.

EXECUTIVE SUMMARY

Blaize Security team has received a set of contracts prepared by the Everstake team. Contracts include:

- PoolB2B.sol - a staking smart contract which allows users to deposit ETH, which is then staked by a specific validator on beacon chain.
- ValidatorList.sol - library, which simplifies the work with the list of validators.

The goal of the audit was to ensure the correctness of interaction with Beacon chain deposit smart contract, validate that smart contracts are optimized in terms of gas usage, and Solidity best practices, validate smart contracts against the list of common vulnerabilities.

There were several low and lowest issues found during the manual audit. Low issues described the unused fee variable and unused functions, while the lowest issues were connected to gas optimization, validation of logic, and other improvements of smart contracts. Everstake team has successfully fixed or verified all of the issues found. Additionally, auditors have proposed several gas optimizations in order to decrease gas costs of functions. All the issues and proposed optimizations can be seen in Complete analysis section.

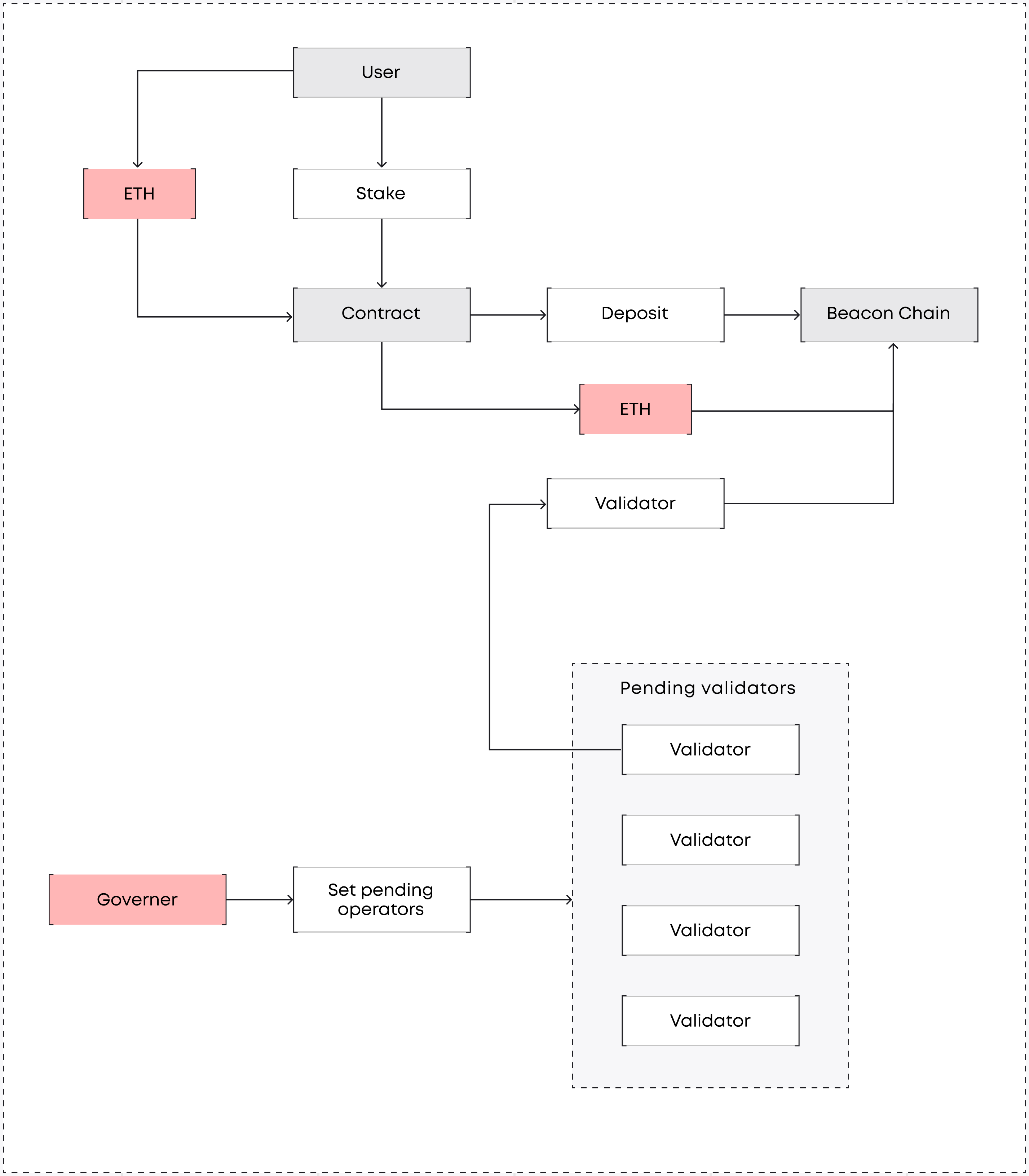
Blaize Security team has also prepared a set of fork-tests in order to validate the correctness of smart contract's logic and interaction with Beacon Chain deposit smart contract.

The overall security of smart contracts is high enough. Contracts are well-written, contain a natspec documentation, and a gas-optimized.

	RATING
Security	10
Gas usage and logic optimization	10
Code quality	10
Test coverage**	10
Total	10

**Contract has a native coverage prepared by the Everstake team, though Blaize Security has prepared their own set of unit tests and additional scenarios to cover the whole code. The mark shows the final testable code.

E V E R S T A K E


User
ETH
Stake
Contract
Deposit
Beacon Chain
ETH
Validator
Pending validators
Validator
Validator
Validator
Validator
Governer
Set pending operators

COMPLETE ANALYSIS

LOW-1	✓ Verified

Fee is not used

PoolB2Bl.sol: _poolFee variable, FEE_DENOMINATOR constant, setFee(), fee()
Fee functionality is not used in the smart contract. For now, it is just a number stored onchain. So it is either unfinished functionality or it is obsolete functionality.
Auditors also suppose this value may be used in dApp or periphery contracts. Therefore the issue is marked as Low since it needs additional information from the team.

Recommendation:
Add fee functionality usage, OR remove unused functionality OR verify that the fee storage is required for the dApp/other contracts.

Post-audit:
Everstake team has verified that fee is necessary for the Dapp and customers.

LOW-2	✓ Resolved

Unused function

PoolB2Bl.sol: _safeEthSend()
Internal function _safeEthSend() is not used anywhere in the contract. It is a sign of either unfinished or obsolete functionality.

Recommendation:
Consider removing of the unused function

Post-audit:
Function was removed.

LOWEST-1 ✓ Resolved

Variables can be defined as immutable

PoolB2Bl.sol: _depositContact, _withdrawAuthority

These variables get value just once during the deployment of the contract. Therefore it is recommended to mark them as immutable.

Recommendation:

Set variables as immutable.

LOWEST-2 ✓ Resolved

Unnecessary check

PoolB2Bl.sol: _stake()

The function contains a check if the value equals 0. Check is redundant, since _stake() is called in a single place in the stake() function, and the general stake function already contains a check for value to be greater than BEACON_AMOUNT.

Recommendation:

Remove redundant check.

LOWEST-3 ✓ Verified

Matching of the staker with the prepared validator
PoolB2Bl.sol: _stake()

By documentation, Everstake devops team prepares production-ready workers for validators and prepares all necessary credentials (pubkey, signatures, withdrawal params, etc.). Also, the team puts prepared validators info into the contract (via setPendingValidators()). By then, all validators are prepared and dedicated for the user (including multisigs setup). After that, the user may use stake() function from his wallet and provide a deposit to the Beacon contract using the Everstake set of validators. Though, the issue here is that these 2 events (validators preparation and stake by user) are not atomic (which is logical) and are not crosschecked. So, there is no guarantee that the user, through the stake() function, will receive his dedicated set of validators. So we can imagine a scenario of 2 users which will provide a stake, not in the order of how validators are added. So it is recommended to provide validation during the stake so that the user receives his dedicated validator.

The issue is marked as info as it relates to the business logic. Also, by the time of the audit, the exact procedure of connection staker to his dedicated validators is unknown.

Recommendation:

Provide additional checks that the user (staker) will use his dedicated set of validators during the stake() (check by the wallet address or signature check, for example) OR verify that there will be no conflicts from this point of view and Everstake will handle queuing of users through the dApp.

From client:

Everstake team verified that the issue is not actual, as the order of validators is not relevant.

LOWEST-4	✓ Resolved

Lack of documentation.

No information about the contract, functions, storage variables, and other entities. There is minimal commentary. It is recommended to describe all entities with NatSpec comments according to the Solidity style guide in order to ensure clear and user-friendly usage.

Recommendation:

Use NatSpec commentary for all the entities.

Post-audit: Function descriptions were added.

LOWEST-5 ✓ Resolved

Inaccurate version pragma.
`pragma solidity ^0.8.0;` is used in `PoolB2B`. The contract should be deployed with the same compiler version and options with which it has been most tested. Locking the pragma version helps ensure that the contract is not accidentally deployed using a different version. In addition, older versions may contain bugs and vulnerabilities and be less optimized in terms of gas. It is recommended to use the latest version of Solidity and specify the exact pragma.

Recommendation:

Specify the latest version of Solidity in the pragma statement.

Post-audit:
'pragma solidity 0.8.19' is used now.

LOWEST-6 ✓ Resolved

Boolean equality.
PoolB2BI.sol: setPendingValidators(), line 172.
There is the comparison to `false`. Boolean constants can be used directly and do not need to be compared to `true` or `false`.

Recommendation:

Remove the equality to `false`.

LOWEST-7 ✓ Resolved

Unnecessary private visibility.
PoolB2Bl.sol: _poolBalance, _poolFee, _withdrawAuthority, _depositContract, _governor.
The contract contains storage variables with private visibility. Usually, private visibility for state variables is worth using in case of inheritance or insufficiency of the default public getter (e.g., in case of a structure including an array) or special rules for such a function, including conversions, requirements, and the like. Whereas getters for listed variables do not apply to these cases. Such use complicates the code and reduces readability.

Recommendation:

Use public visibility for these variables.

LOWEST-8 ✓ Resolved

Removed validator can't be added again.
PoolB2B.sol: _removePendingValidator(), setPendingValidators().
When the validator is removed, info about him in mapping `_validatorsRegistry`is not removed. Thus, the value for a removed validator is stayed as true. This might be confusing for the Dapp in case protocol validates if the validator is registered by using this mapping. Also, once the validator is removed, it is impossible to add him to the array of pending validators again due to validation in setPendingValidators(), line 172.

Recommendation:

Delete a value from mapping `_validatorsRegistry`for the removed validator **OR** verify that the value should not be deleted, and once the validator is removed, it can't be added to pending validators again.

Post-audit: Removed validators can be added again now.

LOWEST-9 ✓ Resolved

Confusing naming.
PoolB2B.sol.

- The mapping `_validatorsRegistry`is used as a validator usage mark, not as a registry. (See Info-8 for details).

Recommendation.
Rename it according to usage.

- The function `deposit()`has the private visibility, but named without an underscore '_', unlike other private functions. This is misleading.

Recommendation.
Agree on the naming by adding an underscore.

LOWEST-10 ✓ Resolved

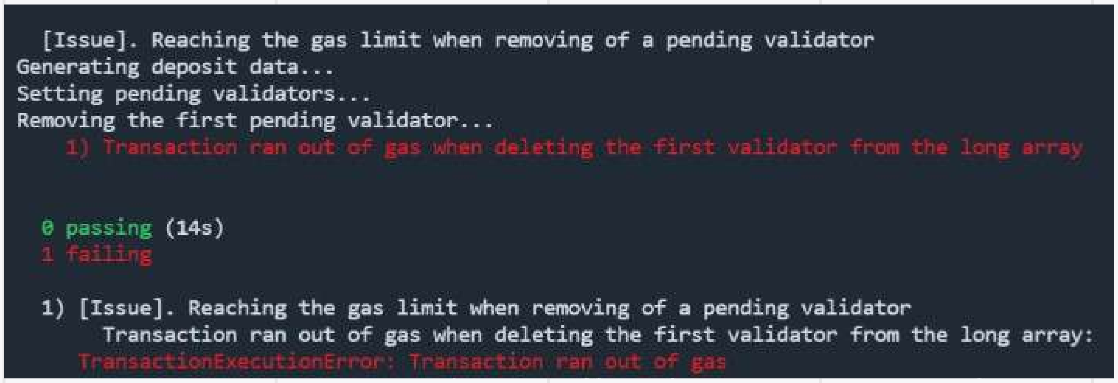
Lack of getters.
PoolB2B.sol: BEACON_AMOUNT, _validatorsRegistry, FEE_DENOMINATOR, ETH1_ADDRESS_WITHDRAWAL_PREFIX, _pendingValidators.
There are no getters for the listed state variables and constants. The lack of getters makes it difficult to keep track of the storage state and debug issues that arise, as well as reduces usability. In case of `_pendingValidators`, there is only getter for a `pubkey`, but not for the others. Note that the default public getter does not include dynamically-sized byte arrays, so it is worth creating it manually. There is no point in making these values private, as input parameters can be viewed by transaction, and constant values can be viewed in the block explorer after verification or by decompiling byte-code otherwise.

Recommendation.
Add getters for the variables.

LOWEST-11 ✓ Resolved

The issue about out-of-gas transaction when removing the first pending validator

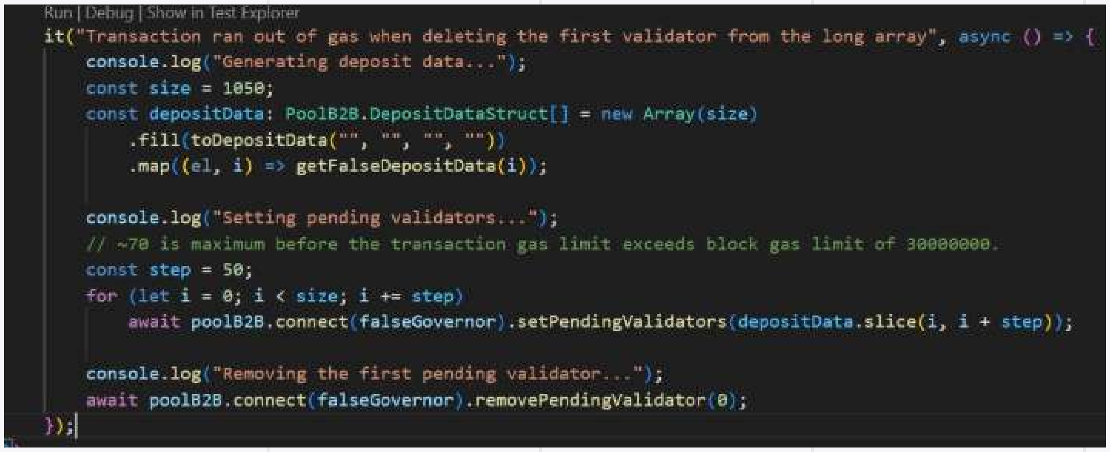
(1). It is also worth noting that if there are about 1050 pending validators in the array, the transaction would not be feasible at all, as it will run out of gas. Due to the swapping of a lot of elements on lines 189–191.



```
  [Issue]. Reaching the gas limit when removing of a pending validator
Generating deposit data...
Setting pending validators...
Removing the first pending validator...
    1) Transaction ran out of gas when deleting the first validator from the long array


  0 passing (14s)
  1 failing

  1) [Issue]. Reaching the gas limit when removing of a pending validator
       Transaction ran out of gas when deleting the first validator from the long array:
     TransactionExecutionError: Transaction ran out of gas
```



```
Run | Debug | Show in Test Explorer
it("Transaction ran out of gas when deleting the first validator from the long array", async () => {
    console.log("Generating deposit data...");
    const size = 1050;
    const depositData: PoolB2B.DepositDataStruct[] = new Array(size)
        .fill(toDepositData("", "", "", ""))
        .map((el, i) => getFalseDepositData(i));

    console.log("Setting pending validators...");
    // ~70 is maximum before the transaction gas limit exceeds block gas limit of 30000000.
    const step = 50;
    for (let i = 0; i < size; i += step)
        await poolB2B.connect(falseGovernor).setPendingValidators(depositData.slice(i, i + step));

    console.log("Removing the first pending validator...");
    await poolB2B.connect(falseGovernor).removePendingValidator(0);
});
```

Post-audit.

Everstake team has rewritten the algorithm, so that stack is used now. Thus it is no longer needed to iterate through all validators.

LOWEST-12 ✔ **Resolved**

Optimization suggestion for Everstake and out-of-gas transaction issue

Currently, when `setPendingValidators()` is called, an element with three dynamic arrays (`bytes`) is written to the `_pendingValidators` array. The user then, when `stake()` is called, copies such an element to memory at his own expense (`DepositData memory validator` on line 92), removes it from the `pendingValidators` array (`_pendingValidators.pop();` on line 193), and then writes the public key back into the storage (`_validators.push(pubkey);` on line 133). It is worth noting that if the queue of pending validators contains not a couple of elements, but several dozen or more, the cost increases noticeably due to swaps (line 190).

Possible minimum optimization

In addition to what is described above, the user has to do this swap (lines 189–191) multiple times, because instead of doing it once, the first one (`_removePendingValidator(0);` on line 93) is deleted multiple times.

It is worth doing this once, starting the swap with the last pending validator for the current stake when deleting.

Although this will reduce consumption, it still does **not** solve the issue (1) of the transaction going over the gas limit.

More advanced optimization (recommended)

Instead of all this, it is worth:

- replace the two arrays with one mapping;
- add two numbers – the index of the first pending validator and the number of pending validators;
- add a numeric status to the validator data, indicating that it is in the queue, used when staking, or excluded using `removePendingValidator();`

- (optionally) if arrays of used and pending validators are to be obtained, getters can be added, which will collect them by the mentioned mapping since `view` methods do not require a gas payment.

Thus, there will be no need to remove validators from the storage in `_removePendingValidator()` and write the dynamic array `bytes` (public key) again in `deposit()`. Gas consumption will be reduced, especially in cases with a lot of pending validators. It can be considered that issue (1) will be solved. At the same time, more historical data will be left.

That is, it is worth replacing the two arrays
`bytes[] private _validators;` and
`DepositData[] private _pendingValidators;`

with the one mapping

```
// A validator index -> Validator data.
`mapping(uint256 => ValidatorData) public validators;`,
```

where `ValidatorData` is an augmented `DepositData`:

```
enum ValidatorStatus {
    NonExistent,
    Pending,        // Added with `setPendingValidators()`.
    Staked,         // Used ("deleted") at `stake()`.
    Removed         // Excluded ("deleted") with
`removePendingValidator()`.
   }
   struct ValidatorData {
    ValidatorStatus status;

    bytes pubkey;
    bytes withdrawal_credentials;
    bytes signature;
    bytes32 deposit_data_root;
   }.
```

At the same time, store two numbers: the index of the first pending validator in the `validators` mapping and the number of pending validators.

Use the number to check that there are enough validators for `stake()` and subtract from it in the same call. And also use it to add new ones.

Start "deleting" (changing state to `ValidatorStatus.Staked`) from the index of the first pending validator, increasing the index to the new first pending validator.

If there are deleted validators after the index of the first pending validator due to calls to `removePendingValidator()`, then they have `status == ValidatorStatus.Removed` and the index is further increased, skipping them. And there will still be enough validators, because the number is pre-checked.

Also, it is cheaper to use the `nonReentrant` modifier (OpenZeppelin) for `stake()` than to copy each validator into memory (`DepositData memory validator`, line 92) before deleting. That is, use `ValidatorData storage validator` with `nonReentrant`.

This reduces the required computation for `stake()` and `removePendingValidator()`, and it can also be considered that it solves the cost and gas limit issue of swapping and deleting pending validators. Leaves more historical data at the same time.

If you want to get arrays of used ("staked") and pending validators, getters can be made which return the needed arrays by mapping. For used validators before the index of the first pending validator, and for pending validators after. Having regard to `ValidatorStatus.Staked` and `ValidatorStatus.Pending` respectively.

Post-audit.
The array was correctly rewritten using index and overwrite instead of pushing in the similar way as suggested. Although, a mapping was not used instead of an array. However, this is not considered a disadvantage, as it depends on the specifics of the task and individual preferences of the developer. In addition, a library was written to handle such an array in a convenient way.

LOWEST-13 ✓ **Resolved**

Unnecessary requirements.
PoolB2B.sol.

- `_stake()`.On line 106, an array of validators is required to contain pending validators (`require(isPresented, 'Pending validator');`). However, before this on line 101, the requirement uses the `.length()` method, which takes into account the number of active (pending) validators, eliminating such a possibility. Thus, it is possible not to take the value of `isPresented` locally and to require it to be true in this case.

Recommendation. Do not take the value of `isPresented` locally and not require it to be true in this case.

- `_deposit()`.

Due to what is described in point 1, the requirement on line 142 (`_knownValidators[validatorHash] == ValidatorStatus.Pending`) also is unreachable, since any validator returned on line 105 will have the pending status.
Recommendation. Remove it.

- `_removePendingValidator()`.

On line 224 it is also required `_knownValidators[validatorHash] == ValidatorStatus.Pending`, but before that, on line 220 it is checked a requirement similar to the one in point 1 (line 101). That is, it also turns out to be unreachable because validators with a different status are cut off using the `.length()` method.
Recommendation. Remove it.

LOWEST-14 ✓ **Resolved**

Unnecessary gas consumptions.
ValidatorList.sol: `remove()`
There is passed an array of deposit data in `setPendingValidators()`, but they are added to the list one at a time, and the index is updated for each. It is recommended to add method `addBatch(List storage set, DepositData[] calldata validators[])` to `ValidatorList` to change `_activeElementIndex` once in `setPendingValidators()`, adding the passed validators at a time. How many will need to be added via `.push()` will also only be checked once.

Recommendation.
Optimize this by adding the method.

Post-audit.
Tested that there is little difference due to additional indexes during array iterating.

		PoolB2B.sol ValidatorList.sol
✓	Re-entrancy	Pass
✓	Access Management Hierarchy	Pass
✓	Arithmetic Over/Under Flows	Pass
✓	Delegatecall Unexpected Ether	Pass
✓	Default Public Visibility	Pass
✓	Hidden Malicious Code	Pass
✓	Entropy Illusion (Lack of Randomness)	Pass
✓	External Contract Referencing	Pass
✓	Short Address/ Parameter Attack	Pass
✓	Unchecked CALL Return Values	Pass
✓	Race Conditions / Front Running	Pass
✓	General Denial Of Service (DOS)	Pass
✓	Uninitialized Storage Pointers	Pass
✓	Floating Points and Precision	Pass
✓	Tx.Origin Authentication	Pass
✓	Signatures Replay	Pass
✓	Pool Asset Security (backdoors in the underlying ERC-20)	Pass

CODE COVERAGE AND TEST RESULTS FOR ALL FILES, PREPARED BY EVERSTAKE TEAM

Contract: Pool_B2B

- ✓ success: `deposit`(single user, stake completely used) (3646ms)
- ✓ success: `deposit`(single user with 2 deposits amount, stake completely used) (6406ms)
- ✓ success: `deposit`(multi user, stake completely used) (6735ms)
- ✓ success: `remove pending validator`(4438ms)
- ✓ fail: `remove pending validator not governor call` (2459ms)
- ✓ fail: `outworld deposit
- ✓ fail: `same validator in batch`(2153ms)
- ✓ fail: `wrong deposit amount, GT than BN_BEACON
- ✓ fail: `wrong deposit amount, LT than BN_BEACON`
- ✓ fail: `set pending validators`(not governor caller)
- ✓ fail: `stake`(not enough pending validators) (180ms)
- ✓ fail: `multi stake`(not enough pending validators) (3635ms)
- ✓ fail: `set pending validators`(wrong withdraw creds)
- ✓ fail: `deposit`(wrong deposit creds) (2426ms)
- ✓ fail: `alredy added pending validator`(same validator) (2302ms)
- ✓ fail: `alredy added validator`(same validator) (3603ms)
- ✓ fail: `wrong pubkey len`(less than)
- ✓ fail: `wrong pubkey len`(gt than)
- ✓ fail: `wrong sigrature len`(less than)
- ✓ fail: `wrong sigrature len`(gt than)

20 passing (3m)

TEST COVERAGE RESULTS

TOKENDEAL TEAM

FILE	% STMTS	% BRANCH	% FUNCS
OperatorFilterer.sol	40	30	57.14
Holoself.sol	78.95	45.24	53.85

CODE COVERAGE AND TEST RESULTS FOR ALL FILES, PREPARED BY BLAIZE SECURITY TEAM

PoolB2B

Deployment

✓ Initializes (526ms)

Actions

Setting of pending validators

✓ Sets a pending validator (42ms)

✓ Reverts when setting if wrong withdrawal credentials

✓ Reverts when setting if a wrong length of a public key

✓ Reverts when setting if a wrong length of a signature

✓ Reverts when setting if a validator has already been added

✓ Prevents non-governors from setting

Staking

✓ Stakes (50ms)

✓ Stakes by two users [skip-on-coverage] (74ms)

✓ Reverts when staking if a beacon amount is not a multiple of 32 Ether

✓ Reverts when staking if a zero beacon amount

✓ Reverts when staking if not enough pending validators

Removal of pending validators

✓ Removes the first pending validator

✓ Removes the first pending validator when there are two

✓ Removes the second pending validator when there are three [skip-on-coverage] (41ms)

✓ Removes the last pending validator when there are three [skip-on-coverage] (41ms)

✓ Reverts when removing if an out-of-range index

✓ Prevents non-governors from removing

Setting of parameters

✓ Sets the governor's address

✓ Prevents non-governors from setting the governor's address

✓ Sets the pool fee

✓ Returns false`and does not set a new pool fee if the passed value exceeds 100%

✓ Prevents non-governors from setting the pool fee

```
# Getting
✓ Gets staker's balance (43ms)
✓ Gets the pool balance (43ms)
✓ Gets the pool fee
✓ Gets an address of the withdrawal authority
✓ Gets an address of the deposit contract
✓ Gets the number of validators (45ms)
✓ Gets validator's public key (46ms)
✓ Reverts when getting validator's public key if an
  out-of-range index
✓ Gets the number of pending validators
✓ Gets a public key of a pending validator
✓ Reverts when getting a public key of a pending
  validator if an out-of-range index
✓ Gets an address of the governor

35 passing (7s)
```

TEST COVERAGE RESULTS

BLAIZE SECURITY TEAM

FILE	% STMTS	% BRANCH	% FUNCS
PoolB2B.sol	94	91.18	95.45
All files	94	91.18	95.45

The team also performed additional round of fork testing (with the ETH stake contract) and manual testing to ensure the correctness of logic.

DISCLAIMER

The information presented in this report is an intellectual property of the customer including all presented documentation, code databases, labels, titles, ways of usage as well as the information about potential vulnerabilities and methods of their exploitation. This audit report does not give any warranties on the absolute security of the code. Blaize.Security is not responsible for how you use this product and does not constitute any investment advice.

Blaize.Security does not provide any warranty that the working product will be compatible with any software, system, protocol or service and operate without interruption. We do not claim the investigated product is able to meet your or anyone else requirements and be fully secure, complete, accurate and free of any errors and code inconsistency.

We are not responsible for all subsequent changes, deletions and relocations of the code within the contracts that are the subjects of this report.

You should perceive Blaize.Security as a tool that helps to investigate and detect the weaknesses and vulnerable parts that may accelerate the technology improvements and faster error elimination.