Veridise

Auditing Report

Hardening Blockchain Security with Formal Methods

FOR

DendrETH

Veridise Inc.

July 1, 2024

► **Prepared For:**

Metacraft Labs

► **Prepared By:**

Ian Neal
Kostas Ferles

► **Contact Us:** contact@veridise.com

► **Version History:**

July 1, 2024	V1
June 18, 2024	Initial Draft

Contents

1 Executive Summary

From May 31, 2024 to June 14, 2024, Metacraft Labs engaged Veridise to review the security of their DendrETH project. Veridise conducted the assessment over 4 person-weeks, with 2 engineers reviewing code over 2 weeks from commits `8c8e6d7 - 1242778`. The auditing strategy involved a tool-assisted analysis of the source code performed by Veridise engineers as well as extensive manual code review.

Project summary. In its core, DendrETH implements the beacon chain light client syncing algorithm in the form of a smart contract for multiple targeted blockchains, aiming to enable the creation of secure cross-blockchain bridges that do not require a trusted operator via the use of zero-knowledge circuits. In contrast to previous ZK implementations of light clients, DendrETH leverages EIP-4788 that exposes the beacon block root in the EVM.

Veridise's security assessment covered the DendrETH beacon chain light client zero-knowledge circuits, implemented in Plonky2, and some smart contracts integrating the circuits with two protocols, namely Lido and Diva. Both of these protocols are liquid staking protocols that need an oracle to provide information about their validators. Integrating these protocols with a ZK oracle, as implemented by this project, minimizes their trust assumptions.

Code assessment. The DendrETH developers provided the source code of the DendrETH contracts and circuits for review. The source code appears to be mostly original code written by the DendrETH developers. It contains some documentation in the form of READMEs and documentation comments on functions and storage variables. To facilitate the Veridise auditors' understanding of the code, the DendrETH developers provided some additional documents that described the system architecture as well as documents that described the integration process with Lido, one of the two liquid staking protocols that intend to use DendrETH. Additionally, the Veridise team also met frequently with the Metacraft Labs team to resolve any questions related to the DendrETH implementation.

Out-of-scope code. One notable thing about this audit was the presence of several non-audited dependencies. One of these dependencies was a macro library developed by the Metacraft Labs team that abstracts away some boilerplate related to writing Plonky2 circuits. Other dependencies were third-party libraries that implement cryptographic primitives in Plonky2. The Veridise auditors studied such dependencies thoroughly and made sure they understood the proper way of using these libraries. Veridise auditors did not review the out-of-scope code for bugs or undocumented assumptions, and assumed the functions were implemented correctly for the purposes of this review.

Test suite. The source code contained a test suite, which the Veridise auditors noted that was covering most security-critical components of the system. Additionally, the Veridise auditors also noticed several negative test cases for critical circuits of the system. However, the auditors would recommend extending this test suite with more tests for all components of the system and adding even more negative test cases, especially for components that handle security-critical data like tree roots.

Overall, the Veridise team found the organization of the codebase clean and easy to follow. Also, during the meetings with the Metacraft Labs team, the Veridise auditors noticed that they are a very security-focused team when developing systems.

During the audit, the Metacraft Labs developers made several functional changes to the code. This change occurred due to changes in the requirements from one of the two liquid staking protocols that wanted to integrate with DendrETH. However, this did not have a major impact on the audit, as the changes occurred early in the audit and the scope of the changes were well-communicated to the Veridise auditors.

Summary of issues detected. The audit uncovered 16 issues, 1 of which is assessed to be of high or critical severity by the Veridise auditors. Specifically, the high-severity issue could potentially DoS the Diva integration smart contract by calculating the wrong accumulator for the current set of validators. The Veridise auditors also identified 3 low-severity issues, 7 warnings, and 5 informational findings. The DendrETH developers have acknowledged all the issues and are currently working on resolving them.

Recommendations. After auditing the protocol, the auditors had a few suggestions to improve the DendrETH.

Public inputs handling. As mentioned earlier, the Metacraft Labs team has developed a Rust macro library to abstract away some boilerplate Plonky2 code. However, for some public inputs the developers chose to manually write the part of the Plonky2 code that is typically handled by the macro library. The Veridise auditors found this choice confusing and would recommend handling all such cases uniformly. This will increase the readability of the code as well as avoid any maintainability issues for future versions of the code.

Un-audited dependencies. We would also recommend that the Metacraft Labs team eventually move away from un-audited dependencies. For some cases this can be achieved by writing in-house components for small modules. For larger dependencies, the team would need to either find an audited alternative or order their own audit on the dependency. The Metacraft Labs team has already taken steps in both of these directions.

Document implicit assumptions. While reviewing the ZK circuits, the Veridise noticed that they work under some implicit and undocumented assumptions. As an example, refer to the impact section of issue V-DNDR-VUL-005 that describes such a scenario. We would recommend clearly documenting all such assumptions to avoid any bugs in client code.

Magic numbers. Finally, the Veridise auditors also noticed a significant amount of magic numbers while reviewing the code (e.g., generalized Merkle tree indices). We would recommend replacing all these magic numbers with Rust constants and adding appropriate documentation on how they were derived.

Disclaimer. We hope that this report is informative but provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the system is secure in all dimensions. In no event shall Veridise or any of its employees be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the results reported here.

2 Project Dashboard

Table 2.1: Application Summary.

Name	Version	Type	Platform
DendrETH	`8c8e6d7 - 1242778`	Plonky2	Ethereum

Table 2.2: Engagement Summary.

Dates	Method	Consultants Engaged	Level of Effort
May 31 - June 14, 2024	Manual & Tools	2	4 person-weeks

Table 2.3: Vulnerability Summary.

Name	Number	Acknowledged	Fixed
Critical-Severity Issues	0	0	0
High-Severity Issues	1	1	1
Medium-Severity Issues	0	0	0
Low-Severity Issues	3	3	2
Warning-Severity Issues	7	7	7
Informational-Severity Issues	5	5	5
TOTAL	16	16	15

Table 2.4: Category Breakdown.

Name	Number
Maintainability	9
Data Validation	3
Logic Error	1
Access Control	1
Usability Issue	1
Gas Optimization	1

3 Audit Goals and Scope

3.1 Audit Goals

The engagement was scoped to provide a security assessment of DendrETH's Plonky2 circuits and associated Solidity smart contracts. In our audit, we sought to answer questions such as:

- Does the SHA256 implementation match the NIST specification?
- Are the recursive proofs correctly verified?
- Are bit-wise operations performed correctly?
- Do the Solidity smart contracts correctly interact with the Plonky2 circuits?
- Are SSZ (simple serialize) and Merkleization operations performed in accordance with the Ethereum consensus specifications?
- Are all circuit targets properly constrained?

3.2 Audit Methodology & Scope

Audit Methodology. To address the questions above, our audit involved a combination of human experts and automated program analysis & testing tools. In particular, we conducted our audit with the aid of the following techniques:

- *Static analysis.* To identify potential common vulnerabilities, we leveraged our custom smart contract analysis tool Vanguard. Vanguard is designed to find instances of common smart contract vulnerabilities, such as reentrancy and uninitialized variables.
- *Differential fuzzing.* We also leverage differential fuzz testing to determine if the protocol may deviate from the expected behavior. To do this, we implement a custom testing application that compares the output of the function-under-test to a reference implementation. We then use the fuzzing framework to generate fuzzing inputs to determine if there are any inputs where the two implementations deviate in output. We discuss details and results of our fuzzing campaign at the end of the report.

Scope. The scope of this audit is limited to:

- The `beacon-light-client/plonky2/crates/circuits` folder (excluding the `deposits_accumulator_balance_aggregator` and `deposits_accumulator_commitment_mapper` sub-folders) of the source code provided by the DendrETH developers, which contain the zero knowledge circuits of the DendrETH written in Plonky2.
- The `beacon-light-client/solidity/contracts` folder (excluding the `bridge` and `test` sub-folders), which contain the smart contracts of the DendrETH project written in Solidity.

Methodology. Veridise auditors met with the DendrETH team at the beginning of the audit to review the high-level structure of DendrETH. The Veridise auditors also inspected the provided tests and read the provided DendrETH documentation. They then began a manual review of

the code assisted by both static analyzers and automated testing. During the audit, the Veridise auditors regularly met with the DendrETH developers to ask questions about the code.

3.3 Classification of Vulnerabilities

When Veridise auditors discover a possible security vulnerability, they must estimate its severity by weighing its potential impact against the likelihood that a problem will arise. Table 3.1 shows how our auditors weigh this information to estimate the severity of a given issue.

Table 3.1: Severity Breakdown.

	Somewhat Bad	Bad	Very Bad	Protocol Breaking
Not Likely	Info	Warning	Low	Medium
Likely	Warning	Low	Medium	High
Very Likely	Low	Medium	High	Critical

In this case, we judge the likelihood of a vulnerability as follows in Table 3.2:

Table 3.2: Likelihood Breakdown

Not Likely	A small set of users must make a specific mistake
Likely	Requires a complex series of steps by almost any user(s) - OR - Requires a small set of users to perform an action
Very Likely	Can be easily performed by almost anyone

In addition, we judge the impact of a vulnerability as follows in Table 3.3:

Table 3.3: Impact Breakdown

Somewhat Bad	Inconveniences a small number of users and can be fixed by the user
Bad	Affects a large number of people and can be fixed by the user - OR - Affects a very small number of people and requires aid to fix
Very Bad	Affects a large number of people and requires aid to fix - OR - Disrupts the intended behavior of the protocol for a small group of users through no fault of their own
Protocol Breaking	Disrupts the intended behavior of the protocol for a large group of users through no fault of their own

Vulnerability Report 4

In this section, we describe the vulnerabilities found during our audit. For each issue found, we log the type of the issue, its severity, location in the code base, and its current status (i.e., acknowledged, fixed, etc.). Table 4.1 summarizes the issues discovered:

Table 4.1: Summary of Discovered Vulnerabilities.

ID	Description	Severity	Status
V-DNDR-VUL-001	_binarySearchBlock may return old accumulator	High	Fixed
V-DNDR-VUL-002	Little-endian function returns big-endian bits	Low	Fixed
V-DNDR-VUL-003	Centralization Risk	Low	Acknowledged
V-DNDR-VUL-004	verifier may become out of sync with digest	Low	Fixed
V-DNDR-VUL-005	Implicit assumption on BigUintTarget	Warning	Fixed
V-DNDR-VUL-006	Manual public inputs registration	Warning	Fixed
V-DNDR-VUL-007	Undocumented endianness	Warning	Fixed
V-DNDR-VUL-008	Balance index range not enforced	Warning	Fixed
V-DNDR-VUL-009	Unused program constructs	Warning	Fixed
V-DNDR-VUL-010	Unchecked validatorsCount return value	Warning	Fixed
V-DNDR-VUL-011	Missing address zero-checks	Warning	Fixed
V-DNDR-VUL-012	Naming recommendations	Info	Fixed
V-DNDR-VUL-013	Miscellaneous improvements	Info	Fixed
V-DNDR-VUL-014	Complex constraints for conjunction	Info	Fixed
V-DNDR-VUL-015	Unnecessary state variable	Info	Fixed
V-DNDR-VUL-016	Typos and incorrect comments	Info	Fixed

4.1 Detailed Description of Issues

4.1.1 V-DNDR-VUL-001: _binarySearchBlock may not return the latest accumulator

Severity	High	**Commit**	1242778
Type	Logic Error	**Status**	Fixed
File(s)	`solidity/contracts/validators_accumulator/ValidatorsAccumulator.sol`		
Location(s)	function _binarySearchBlock		
Confirmed Fix At	72c04ec		

Function `ValidatorsAccumulator._binarySearchBlock` performs a binary search in the contract's `snapshots` mapping to find the accumulator for a specific block number, which is part of the struct stored in `snapshots`. However, as shown below, the function terminates at the first discovered entry with a matching block number. This would be correct only if each block can have only one accumulator, which does not appear to be the case since multiple deposits can occur in the same block.

```
while (upper > lower) {
     uint256 index = upper - (upper - lower) / 2; // ceil, avoiding overflow
     DepositData memory snapshot = snapshots[index];
     if (snapshot.blockNumber == blockNumber) {
       return index;
     }
     ...
}
```

Snippet 4.1: Snippet from `_binarySearchBlock()`.

Therefore, the binary search might return a wrong index if multiple accumulators share the same block. For example, assuming the `snapshot` mapping contains the following entries:

```
{
  0 -> {b0, acc0}
  1 -> {b0, acc1}
  2 -> {b0, acc2}
}
```

Snippet 4.2: Example `snapshot` mapping with multiple accumulators sharing the same block `b0`.

Calling `_binarySearchBlock` with `b0` will return the second entry `{b0, acc1}` instead of the third one as one would have expected.

Impact If users of `BalanceVerifierDiva.verify` construct a proof for all validators that had made a deposit in the current block, this can potentially cause a denial-of-service (DoS) since `BalanceVerifierDiva.verify` relies on the accumulator returned by `ValidatorsAccumulator.findAccumulatorByBlock` (which uses `_binarySearchBlock` internally). This can be bypassed in cases where the `_binarySearchBlock` does not return the expected result by simply creating a proof that does not include the validators that deposited within the same block but were not

part of the returned accumulator. However, this requires low-level knowledge of the system's implementation and it will significantly deteriorate user experience.

Recommendation We recommend finding the highest index with the same block number in `snapshots` and returning this accumulator instead. Alternatively, you can make the key of the `snapshots` mapping to be the block number, in which case there is no need for a binary search.

Developer Response The developers attempted to fix the issue by modifying the contract logic so that the key of the `snapshots` mapping is the block number.

Veridise Response `BalanceVerifierDiva.verify` now takes the `blockNumber` parameter as a `uint64`, but the `block.number` state variable is specified to be a `uint256`. This truncation is unnecessary and could led to potential issues. We recommend reverting this change and continuing to use `uint256` as the `blockNumber` type.

Developer Response The circuits require that `blockNumber` be a `uint64` in order for the on-chain verification to pass.

Veridise Response We acknowledge this requirement. We recommend adding some documentation to the code base to explain this requirement for future readers, but we will consider this issue fixed as this block number truncation should not cause any issues for the foreseeable future.

4.1.2 V-DNDR-VUL-002: Little-endian function returns big-endian bits

Severity	Low	**Commit**	1242778
Type	Maintainability	**Status**	Fixed
File(s)	plonky2/crates/circuits/src/utils/circuit/mod.rs		
Location(s)	fn target_to_le_bits		
Confirmed Fix At	2a1e979		

The function `target_to_le_bits` indicates that returns the target in little-endian bit order, but the logic of the function indicates that the bits will be return in big-endian order.

```
pub fn target_to_le_bits<F: RichField + Extendable<D>, const D: usize>(
    builder: &mut CircuitBuilder<F, D>,
    number: Target,
) -> [BoolTarget; 64] {
    builder
        **.split_le(number, 64)** // VERIDISE: bit order is little-endian
        .into_iter()
        **.rev()** // VERIDISE: bit order is now big-endian
        .collect_vec()
        .try_into()
        .unwrap()
    // VERIDISE: bit order is big-endian at return
}
```

Snippet 4.3: Big-endian logic in `target_to_le_bits()`, annotations and emphasis added.

Impact While all current users of this function are aware that the output is actually in big-endian ordering, future updates may occur without this knowledge. Future users will then receive bits in the reverse order than expected, which could lead to computation errors.

Recommendation Either rename the function to indicate that it returns big-endian bits or remove the `rev()` call so that little-endian bits are returned.

Developer Response The developers have renamed the function to `target_to_be_bits`.

4.1.3 V-DNDR-VUL-003: Centralization Risk

Severity	Low	**Commit**	1242778
Type	Access Control	**Status**	Acknowledged
File(s)	See issue description		
Location(s)	See issue description		
Confirmed Fix At	N/A		

Similar to many projects, Metalab's `BalanceVerifierDiva` and `BalanceVerifierLido` declare an administrator role (via OpenZeppelin's `Ownable` contract module) that is given special permissions. In particular, these administrators are given the following abilities:

- In both contracts, the owner may change the address of the `verifier` contract.
- In `BalanceVerifierDiva`, the owner may change the address of the `accumulator` contract.

Impact If a private key were stolen, a hacker would have access to sensitive functionality that could compromise the project. For example, a malicious owner could deny service by setting the `verifier` contract address to 0, or allow invalid proofs by setting the `verifier` contract address to the address of a malicious verifier.

Recommendation As these are all particularly sensitive operations, we would encourage the developers to utilize a decentralized governance, a key management service (like Cubist), or multi-sig contract as opposed to a single account, which introduces a single point of failure.

Developer Response The contracts will be deployed by Diva, who will manage the keys themselves.

4.1.4 V-DNDR-VUL-004: verifier may become out of sync with verifier digest

Severity	Low	**Commit**	1242778
Type	Usability Issue	**Status**	Fixed
File(s)	`solidity/contracts/balance_verifier/BalanceVerifier.sol`		
Location(s)	function setVerifier, immutable VERIFIER_DIGEST		
Confirmed Fix At	13df17a		

One of the public inputs to the verifier is the verifier's circuit digest. Contracts inheriting from the `BalanceVerifier` accept a single `VERIFIER_DIGEST` during construction, which is immutable, but these contracts allow the `verifier` contract address to be updated by the owner during the life of the contract.

```
abstract contract BalanceVerifier is Ownable, IBalanceVerifier {
  /// @notice the verifierDigest of the plonky2 circuit
  uint256 public immutable VERIFIER_DIGEST;

  // VERIDISE: ...

  function setVerifier(address _verifier) external override onlyOwner {
    verifier = _verifier;
  }

  // VERIDISE: ...
```

Snippet 4.4: Snippet from `BalanceVerifier`

The `VERIFIER_DIGEST` must match the verifier circuit in order for proof to verify.

Impact If the updated `verifier` address points to an updated version of the verifier with a different digest, the `BalanceVerifier` will no longer be able to verify proofs. In order to use an updated `verifier`, the `BalanceVerifier` implementation will need to be re-deployed with a new `VERIFIER_DIGEST`.

Recommendation Change the `VERIFIER_DIGEST` to no longer be immutable and change the `setVerifier` to update both the verifier's address and digest.

Developer Response The developers have implemented the recommendation. `setVerifier` now accepts both the new verifier address and the new verifier digest.

4.1.5 V-DNDR-VUL-005: Implicit assumption on BigUintTarget

Severity	Warning	**Commit**	1242778
Type	Maintainability	**Status**	Fixed
File(s)	Multiple files		
Location(s)	All uses of BigUintTarget		
Confirmed Fix At	4df145f and ca96fb3		

The project implicitly assumes that every `BigUintTarget` fits in 64 bits. Evidence of this assumption manifest whenever a circuit performs an arithmetic operation that involves a `BigUintTarget`. Specifically, all such operations will trim all 32-bit limbs used to represent the big integer except the first two. For instance, whenever there is an addition that involves two big integers the carry is assumed to be zero and is discarded (see snippet below).

```
let mut range_total_value =
    builder.add_biguint(&l_input.range_total_value, &r_input.range_total_value);

// pop carry
range_total_value.limbs.pop();
```

Snippet 4.5: Snippet from `WithdrawalCredentialsBalanceAggregatorInnerLevel::define()`.

Impact Such implicit and undocumented assumptions can affect the security of the project in multiple ways. For example, future developers may assume that they should always trim limbs of a `BigUintTarget` and introduce severe vulnerabilities. Additionally, this assumption can impose further assumptions on its users. Take the above snippet that performs an addition as an example, the addition will only be correct only if there is no carry. This limits the number of inner levels a proof can have, since the addition can eventually produce a non-zero carry.

Recommendation We have the following recommendations to resolve this issue:

1. First, we would recommend documenting all such assumptions.
2. Second, we would recommend to create a wrapper for `BigUintTarget` that performs all necessary trimming.
3. Third, we would recommend implementing a `Uint64Target` struct that is maintained entirely by the team.

Developer Response The developers have implemented their own target to represent 64 bit integers and have refactored the codebase to use the `Uint64Target` target.

Veridise Response The first PR introduces a number of issues:

- beacon-light-client/plonky2/crates/circuit/src/targets/uint/macro.rs:
  - `assert_limbs_are_valid` does not assert that the limbs are valid, but rather computes and returns a boolean expression that indicates if the limbs are valid, without issuing an assertion. The caller of this function, `impl AddVirtualTarget`, ignores the return value of `assert_limbs_are_valid`, so the valid check is not enforced.

    - We recommend either (1) adding the appropriate assertion within this function or (2) renaming this function to indicate it does not issue the assertion (e.g., `compute_limbs_are_valid`), annotating this function with the `[must_use]` macro, and adding the appropriate assertions at the call sites of the function.
- V-DNDR-VUL-005 and `mul` truncate the carry limbs that may arise from their internal `biguint` operations.
    - We recommend either (1) constraining the carry to be zero or (2) documenting this behavior (as callers may need to validate the arguments will not generate a carry).
- In `rem`, the `rem_biguint` limbs may be smaller than `$c`, so the function should iterate over `rem_biguint.num_limbs()` rather than `$c`
- The `make_uint32_n` macro implicitly requires that the size of `$b` in bits is equal to `32 * $c`, as `$c` is the number of `U32Target` limbs used by the defined type (e.g., the `U64Target` type has `$b = u64`, meaning `$c` must equal `2`).
    - The macro is currently being used correctly at all call sites, but we recommend either (1) enforcing this requirement during macro generation so any misuse will prevent compilation or (2) documenting this requirement for future users.

Developer Response

> `assert_limbs_are_valid` does not assert that the limbs are valid, but rather computes and returns a boolean expression that indicates if the limbs are valid, without issuing an assertion. The caller of this function, `impl AddVirtualTarget`, ignores the return value of `assert_limbs_are_valid`, so the valid check is not enforced.

This has been fixed by adding an assertion within `assert_limbs_are_valid`.

> V-DNDR-VUL-005 and `mul` truncate the carry limbs that may arise from their internal `biguint` operations.

The developers note that the `Uint*Target` family of targets are intended to behave like native unsigned integers, hence the implicit truncation is expected. The developers agree that this behavior should be documented. They will also consider implementing arithmetic operations with overflow in the future, but intend to leave the functionality as-is for the time being.

> In `rem`, the `rem_biguint` limbs may be smaller than `$c`, so the function should iterate over `rem_biguint.num_limbs()` rather than `$c`.

This is incorrect since the remainder has the same number of limbs as the denominator, which is always `$c`.

> The `make_uint32_n` macro implicitly requires that the size of `$b` in bits is equal to `32 * $c`, as `$c` is the number of `U32Target` limbs used by the defined type (e.g., the `U64Target` type has `$b = u64`, meaning `$c` must equal `2`).

The macro will be updated so that the information on the number of limbs to use is derived from the type, rather than passed as a separate macro parameter.

Veridise Response In response to the remainder issue:

> In `rem`, the `rem_biguint` limbs may be smaller than `$c`, so the function should iterate over `rem_biguint.num_limbs()` rather than `$c`.

We acknowledge that the library used for unsigned integer arithmetic will allocate the same limbs for the remainder; the intent of our prior comment was to make the code independent of the underlying used library. We will consider this issue resolved without further changes, but recommend documenting the fact that, as currently implemented, the numerator and denominator will have the same number of limbs.

4.1.6 V-DNDR-VUL-006: BalanceVerificationFinalCircuit registers public inputs manually

Severity	Warning	**Commit**	1242778
Type	Maintainability	**Status**	Fixed
File(s)	plonky2/crates/circuits/src/withdrawal_credentials_-balance_aggregator/final_layer.rs		
Location(s)	fn BalanceVerificationFinalCircuit::define		
Confirmed Fix At	4f53745		

Circuit `BalanceVerificationFinalCircuit` registers its public inputs manually instead of using the `target(out)` macro (see snippet below). This comes in contrast with the conventions followed by the rest of the project. This decision appears to stem from the fact that the `target` macro does not support `Vec` as an input type (`public_inputs_hash_bytes`'s type). However, the length of the vector is known is advance and can be converted to a slice, which is supported by the existing macro.

```
let public_inputs_hash_bytes = public_inputs_hash
    .chunks(8)
    .map(|x| builder.le_sum(x.iter().rev()))
    .collect_vec();

builder.register_public_inputs(&public_inputs_hash_bytes);
```

Snippet 4.6: Snippet from `BalanceVerificationFinalCircuit::define()`.

Impact This can introduce vulnerabilities in future iterations of the project if the way of registering public inputs via the macros changes.

Recommendation We recommend to introduce a field for the public input of the circuit and tag it with `[target(out)]`.

Developer Response The developers have changed `public_inputs_hash_bytes` to a fixed-size array and have tagged it with the `target(out)` macro as recommended.

4.1.7 V-DNDR-VUL-007: Undocumented endianness

Severity	Warning	**Commit**	1242778
Type	Maintainability	**Status**	Fixed
File(s)	`plonky2/crates/circuits/src/utils/circuit/{mod.rs, hashing/merkle/ssz.rs}`		
Location(s)	See issue description		
Confirmed Fix At	ca96fb3		

Many of the serialization/bit conversion functions have an implicit input and/or output endianness that is not documented in the source code. For example, `BigUintTarget` is little-endian, but it is unclear if the `bit_target` argument to `bits_to_biguint_target` is in little-endian or big-endian order without reviewing the source code logic.

Impact Without endian information, it is easy to get confused about data format, making it much harder to maintain the code and easier for bugs to occur.

Recommendation We recommend adding appropriate documentation to all functions where endianness could be a concern to clarify function input and output expectations. We also encourage the use of type aliases or specialized structs (e.g., `BigEndianBits`/ `LittleEndianBits` as aliases for `Vec<BoolTarget>`) so that these endian requirements are obvious even without explicit code comments.

Developer Response The fix for V-DNDR-VUL-005 has removed the need for these functions, so they are now left unused.

Veridise Response As mentioned in V-DNDR-VUL-009, we recommend removing all `BigUintTarget` -related functions that are now unused and left in the code base, as the present potential maintainability issues.

Developer Response The functions have now been removed, with the exception of `biguint_target_from_le_bits` (formerly `bits_to_biguint_target`), which is used in an out-of-scope circuit.

4.1.8 V-DNDR-VUL-008: Balance index range not enforced

Severity	Warning	**Commit**	1242778
Type	Data Validation	**Status**	Fixed
File(s)	plonky2/crates/circuits/src/utils/circuit/mod.rs		
Location(s)	fn get_balance_from_leaf		
Confirmed Fix At	ca96fb3		

`get_balance_from_leaf` assumes that `balance_index` is within the range $[0, 4)$. However, the function does not check this, and if `balance_index` is outside of this range, `get_balance_from_leaf` performs computation as if `balance_index == 0`.

```
pub fn get_balance_from_leaf<F: RichField + Extendable<D>, const D: usize>(
    builder: &mut CircuitBuilder<F, D>,
    leaf: &SSZTarget,
    balance_index: BigUintTarget,
) -> BigUintTarget {
    // VERIDISE: balances_in_leaf.len() == 4
    let balances_in_leaf = split_into_chunks(leaf);
    // VERIDISE: if the loop below does not set the selector, this function outputs
    // the default accumulator value, as if balance_index == 0.
    let mut accumulator = ssz_num_from_bits(builder, &balances_in_leaf[0].clone());
    for i in 1..balances_in_leaf.len() {
        let current_index_t = builder.constant_biguint(&BigUint::from(i as u32));
        let current_balance_in_leaf = ssz_num_from_bits(builder, &balances_in_leaf[i
    ].clone());

        // VERIDISE: if balance_index >= 4, this selector will always be disabled.
        let selector_enabled =
            biguint_is_equal_non_equal_limbs(builder, &current_index_t, &
    balance_index);
        accumulator = select_biguint(
            builder,
            selector_enabled,
            &current_balance_in_leaf,
            &accumulator,
        );
    }

    accumulator
}
```

Snippet 4.7: Annotated snippet from `get_balance_from_leaf()`.

Impact Currently all call sites of `get_balance_from_leaf` enforce that $0 \leq$ `balance_index` < 4, so this does not cause any issues. However, any refactoring of these call sites may result in computational errors, as this range requirement is not explicitly documented anywhere.

Recommendation Add an assertion or check in `get_balance_from_leaf` that enforces `balance_index` is within range.

Developer Response The logic of the `get_balance_from_leaf` has been changed so that, if `balance_index` is not in the range $[0, 4)$, `get_balance_from_leaf` will return 0.

Veridise Response The function still carries an implicit assumption that the range of `balance_index` must be checked by the caller. With this new function definition, it is possible to pass an erroneous `balance_index` of value ≥ 4 and receive a valid return value of 0. We therefore recommend either performing the range check on `balance_index` within `get_balance_from_leaf` or adding documentation to the `get_balance_from_leaf` function to inform users that the caller is expected to perform the range check.

Developer Response The function has been documented and now indicates that a range check is expected to be performed by the caller.

4.1.9 V-DNDR-VUL-009: Unused program constructs

Severity	Warning	**Commit**	1242778
Type	Maintainability	**Status**	Fixed
File(s)	See issue description		
Location(s)	See issue description		
Confirmed Fix At	954e511		

Description The following program constructs are unused:

- plonky2/crates/circuit/src/utils/circuit/hashing/merkle/poseidon.rs
  - function `validate_merkle_proof_const_poseidon`
- plonky2/crates/circuit/src/utils/circuit/hashing/merkle/sha256.rs
  - function `assert_merkle_proof_is_valid_sha256`
- plonky2/crates/circuits/src/utils/mod.rs
  - function `u64_to_ssz_leaf`
- plonky2/crates/circuits/src/utils/circuit/mod.rs
  - function `biguint_target_from_limbs`
- solidity/contracts/validators_accumulator/interfaces/IValidatorsAccumulator.sol
  - error `InvalidCaller()`
- solidity/contracts/validators_accumulator/ValidatorsAccumulator.sol
  - function `toLe64`

Impact These constructs may become out of sync with the rest of the project, leading to errors if used in the future.

Developer Response The unused constructs have been removed from the solidity contracts. `u64_to_ssz_leaf` is used in code outside of the audit scope, so it will be kept. The other circuit functions will also be kept as utility functions.

4.1.10 V-DNDR-VUL-010: Unchecked validatorsCount return value

Severity	Warning	**Commit**	1242778
Type	Data Validation	**Status**	Fixed
File(s)	`solidity/contracts/balance_verifier/BalanceVerifierDiva.sol`		
Location(s)	function verify		
Confirmed Fix At	72c04ec		

`BalanceVerifierDiva.verify` calls `findAccumulatorByBlock`, which returns the number of accumulators at the given block (`validatorsCount`) and the accumulator at the given block (`accumulator`). However, in this invocation, the `validatorsCount` return value is ignored, even though the `verify` function receives validator count information from the caller.

```
function verify(
    bytes calldata proof,
    uint256 slot,
    uint256 blockNumber,
    uint64 balanceSum,
    // VERIDISE: provided validator counts
    uint64 _numberOfNonActivatedValidators,
    uint64 _numberOfActiveValidators,
    uint64 _numberOfExitedValidators,
    uint64 _numberOfSlashedValidators
  ) external override {
    // VERIDISE: ignored return value is validatorsCount
    (, bytes32 accumulator) = IValidatorsAccumulator(ACCUMULATOR)
          .findAccumulatorByBlock(blockNumber);

    // VERIDISE: ...
```

Snippet 4.8: Snippet from `verify()`.

Impact Ignoring the `validatorsCount` return value is a missed opportunity to check for errors. For example, the bug described in V-DNDR-VUL-001 could be caught by checking `validatorsCount`, as it would not match the input arguments to `verify`.

Recommendation Add a require statement to check that `validatorsCount` matches the input arguments. As `validatorsCount` is incremented each time that `ValidatorsAccumulator.deposit` is invoked and never decremented, `validatorsCount` should be equal to the sum of all provided validator counts, i.e., `validatorsCount` should equal `_numberOfNonActivatedValidators` + `_numberOfActiveValidators` + `_numberOfExitedValidators` + `_numberSlashedValidators`.

Developer Response The `findAccumulatorByBlock` function has been rewritten and no longer returns a `validatorsCount` value, instead now only returning the `accumulator`.

4.1.11 V-DNDR-VUL-011: Missing address zero-checks

Severity	Warning	**Commit**	1242778
Type	Data Validation	**Status**	Fixed
File(s)	See issue description		
Location(s)	See issue description		
Confirmed Fix At	79c0666		

Description The following functions take addresses as arguments, but do not validate that the addresses are non-zero:

- solidity/contracts/balance_verifier/BalanceVerifier.sol:
  - `constructor()`: `_verifier` is not validated.
  - `setVerifier()`: `_verifier` is not validated.
- solidity/contracts/balance_verifier/BalanceVerifierDiva.sol:
  - `constructor()`: `_accumulator` is not validated.
  - `setAccumulator()`: `_accumulator` is not validated.
- solidity/contracts/validators_accumulator/ValidatorsAccumulator.sol
  - `constructor()`: `_depositAddress` is not validated.

Impact If zero is passed as any of these addresses, the contracts will be unable to function correctly until the address is changed in another transaction (in the case of `_verifier` or `_accumulator`) or until the contract is redeployed (in the case of `_depositAddress`, which becomes immutable). In all cases, these incorrect deployments waste gas.

Developer Response The developers have added `address(0)` checks as recommended.

4.1.12 V-DNDR-VUL-012: Naming recommendations

Severity	Info	**Commit**	1242778
Type	Maintainability	**Status**	Fixed
File(s)	See issue description		
Location(s)	See issue description		
Confirmed Fix At	2a1e979		

Description In the following locations, the auditors identified the following confusing naming conventions:

- plonky2/crates/circuits/src/utils/circuit/mod.rs
  - `target_to_le_bits`: this function actually encodes bits in big-endian format, not little endian format. This function should be renamed, e.g., to `target_to_be_bits`. This issue is discussed in more detail in V-DNDR-VUL-002.
- plonky2/crates/circuits/src/withdrawal_credentials_balance_aggregator/first_level.rs
  - `WithdrawalCredentialsBalanceAggregatorFirstLevel::define`: the variable `will_be_counted` is overloaded three times, each with different meanings.
- plonky2/crates/circuits/src/utils/circuit/hashing/merkle/ssz.rs
  - `ssz_merklelize_bool`: This function performs serialization, so may be better named `ssz_serialize_bool`.
- plonky2/crates/circuits/src/deposit_accumulator_balance_aggregator_diva/first_level.rs
  - `DepositAccumulatorBalanceAggregatorDivaFirstLevel`: the variable `is_valid` is overloaded two times for two different validity checks.

Impact These minor issues may lead to future developer confusion.

Developer Response `target_to_le_bits` has been renamed to `target_to_be_bits`.

Veridise Response We still recommend applying the remaining recommendations.

Developer Response

- `ssz_merklelize_bool` has been deleted in the fix for V-DNDR-VUL-005.
- The two `is_valid` variables have been replaced with `validator_proof_is_valid` and `balance_proof_is_valid`.
- `will_be_counted` will be deleted in an upcoming refactor.

4.1.13 V-DNDR-VUL-013: Miscellaneous improvements

Severity	Info	**Commit**	1242778
Type	Maintainability	**Status**	Fixed
File(s)	See issue description		
Location(s)	See issue description		
Confirmed Fix At	f0f739a		

Description In the following locations, the auditors identified minor issues:

- plonky2/crates/circuits/src/deposits_accumulator_balance_aggregator/first_level.rs
  - `test_deposit_accumulator_leaf_circuit`: unused variable `proof`; rename to `_proof`.
- plonky2/crates/circuits/src/deposits_accumulator_balance_aggregator_diva/inner_level.rs
  - `define` has an empty `where` clause that could be removed.
- plonky2/crates/circuits/src/utils/circuit/hashing/merkle/sha256.rs
  - `assert_merkle_proof_is_valid_const_sha256`: The two statements `let true: builder ._true(); builder.connect(is_valid.target, true.target);` could be simplified to `builder.assert_true(is_valid);`.
- solidity/contracts/balance_verifier/BalanceVerifier.so
  - `_verify`: comment states call is made using `address(this)`, but it is not.

```
// Make the call using 'address(this).call'
(bool success, bytes memory returnData) = verifier.call(
  // Encode the call to the 'verify' function with the public inputs
  abi.encodeWithSelector(PlonkVerifier.Verify.selector, proof, publicInputs)
);
```

Snippet 4.9: Inaccurate comment in `_verify()`.

Impact These minor issues do not affect the correctness of the project, but could be addressed to improve the overall code quality.

Developer Response The developers have implemented the suggested corrections.

4.1.14 V-DNDR-VUL-014: Complex constraints for conjunction

Severity	Info	**Commit**	1242778
Type	Maintainability	**Status**	Fixed
File(s)	`plonky2/crates/circuits/src/utils/circuit/mod.rs`		
Location(s)	fn biguint_is_equal_non_equal_limbs		
Confirmed Fix At	ca96fb3		

Function `biguint_is_equal_non_equal_limbs` checks equality between two big unsigned integers that may have different number of limbs. To do so, it accumulates constraints for every limb in variable `ret` (initialized to `true`) as shown in the snippet below.

```
ret = BoolTarget::new_unsafe(builder.select(limb_equal, ret.target, false_t))
```

Snippet 4.10: Snippet from `biguint_is_equal_non_equal_limbs()`

However, all statements that follow the above pattern can be simplified to `ret = builder.and(ret, limb_equal)`.

Recommendation Consider applying the above re-write.

Developer Response This function is now no longer used due to the refactoring performed to address V-DNDR-VUL-005.

Veridise Response The issue is still present in the code base, even if the function is no longer used. As recommend in V-DNDR-VUL-009, we recommend removing unused functions such as these as they could present issues if they are used in the future.

Developer Response These functions have been deleted, with the exception of `bits_to_biguint_target` (which was renamed to `biguint_target_from_le_bits`), which is used in an out-of-scope circuit.

4.1.15 V-DNDR-VUL-015: Unnecessary state variable

Severity	Info	**Commit**	1242778
Type	Gas Optimization	**Status**	Fixed
File(s)	`solidity/contracts/validators_-accumulator/ValidatorsAccumulator.sol`		
Location(s)	State variable startIndex		
Confirmed Fix At	72c04ec		

Description State variable `startIndex` in contract `ValidatorsAccumulator` is never updated upon construction. Effectively, the state variable can be replaced with zero.

Impact Function `_binarySearchBlock`, which used this state variable, will incur unnecessary gas costs for reading `startIndex`.

Recommendation Consider removing this state variable and replace its occurrence with zero.

Developer Response `startIndex` has been removed.

4.1.16 V-DNDR-VUL-016: Typos and incorrect comments

Severity	Info	**Commit**	1242778
Type	Maintainability	**Status**	Fixed
File(s)	See issue description		
Location(s)	See issue description		
Confirmed Fix At	fa582b5		

Description In the following locations, the auditors identified minor typos and potentially misleading comments and error messages:

- solidity/contracts/balance_verifier/BalanceVerifier.sol:
  - function `_findBlockRoot`: documentation comment contains typo "avaliable" → "available"
- plonky2/crates/circuits/src/serializers.rs
  - function `serde_bool_array_to_hex_string_nested::deserialize`: the `expecting` message should be "a sequence of hex strings" not "a sequence of sequences of 0s or 1s".

```
pub fn deserialize<'de, D, const N: usize, const M: usize>(
        deserializer: D,
    ) -> Result<Array<Array<bool, M>, N>, D::Error>
    where
        D: Deserializer<'de>,
    {
        struct MultipleHexStringsVisitor<const N: usize, const M: usize>;

        impl<'de, const N: usize, const M: usize> Visitor<'de> for
    MultipleHexStringsVisitor<N, M> {
            type Value = Array<Array<bool, M>, N>;

            fn expecting(&self, formatter: &mut fmt::Formatter) -> fmt::Result {
                // VERIDISE: this message is copied from the nested bool vec
    deserializer
                formatter.write_str("a sequence of sequences of 0s or 1s")
            }
    // VERIDISE: ...
```

Snippet 4.11: Snippet from `deserialize()`.

Impact These minor errors may lead to future developer confusion.

Developer Response The typos have been corrected.

5 Fuzz Testing

5.1 Methodology

Our goal was to fuzz test DendrETH to assess the correctness of its SHA256 implementation. We used AFL.rs as our fuzzing framework and wrote a custom test driver that compared the output of DendrETH's SHA256 implementation against the output of a Rust port of OpenSSL's verified SHA256 implementation. We then fuzzed the test driver to determine whether or not there were any inputs that could lead to deviations between the client's SHA256 implementation and the reference implementation—if so, this would indicate that DendrETH's SHA256 implementation was incorrect.

5.2 Results

We performed differential fuzzing on an 8-core machine for 24 hours (1440 minutes) and found no inputs where the client's implementation differed from the reference implementation. We therefore found no issues with the SHA256 implementation used by DendrETH.

Glossary

AFL.rs A rust interface into AFL-plus-plus (https://github.com/AFLplusplus/AFLplusplus). See https://github.com/rust-fuzz/afl.rs/ to learn more. 29

Fast Reed-Solomon IOPPs A transparent, succinct argument to prove [1, 2] a committed function is near a polynomial of low-degree. This is frequently used to produce a polynomial commitment scheme. See https://vitalik.ca/general/2017/11/22/starks_part_2.html for more details . 31

FRI Fast Reed-Solomon IOPPs. 31

PLONK An arithmetization strategy for zero-knowledge circuits developed in [3]. See [4] or https://vitalik.ca/general/2019/09/22/plonk.html for more details. 31

PLONKish A catch-all term for arithmetization strategies for zero-knowledge circuits based off of PLONK, but slightly generalized (typically to handle more gates or a different polynomial commitment). For more details, see https://docs.zkproof.org/pages/standards/accepted-workshop3/proposal-turbo_plonk.pdf. 31

Plonky2 A PLONKish arithmetization developed by Polygon Zero with various optimizations and FRI as the polynomial commitment scheme. See https://github.com/0xPolygonZero/plonky2/ for more details. 1

smart contract A self-executing contract with the terms directly written into code. Hosted on a blockchain, it automatically enforces and executes the terms of an agreement between buyer and seller. Smart contracts are transparent, tamper-proof, and eliminate the need for intermediaries, making transactions more efficient and secure. 1

zero-knowledge circuit A cryptographic construct that allows a prover to demonstrate to a verifier that a certain statement is true, without revealing any specific information about the statement itself. See https://en.wikipedia.org/wiki/Zero-knowledge_proof for more . 1, 31