eOracle Target Contracts v2 (delta)

Smart Contract Security Assessment

January 24, 2025

ABSTRACT

Dedaub was commissioned to perform a security audit of eOracle's target contracts. The audit covers the changes included in a specific PR and only the delta of the changes were reviewed as part of this audit. We have previously performed another audit on part of the codebase which can be found here (eOracle Target Contracts - June 21, 2024). The changes primarily concerned the addition of the ability to pause the `EOFeedManager` contract, a major optimization for the BLS signature verification algorithm along with other minor changes.

eOracle's BLS implementation is based on Hubble's implementation. Previously, the protocol used public keys that lived in the G2 group. However, since the addition operation on this group is expensive to be made on-chain, a more optimized version uses a pair of public keys `(pk1,pk2)` in `(G1,G2)` such that the aggregation of `pk2` from multiple signers is done off-chain, while the aggregation of `pk1`is done on chain. The description of this method can be found here: Optimized BLS multisignatures on EVM - Geometry. This approach allowed for a more efficient way for verifying aggregated BLS signatures as it reduces the computations performed on-chain thus reducing gas consumption. No major issues were found, but only some small inconsistencies in the BLS refactoring were spotted which were correctly addressed by the team.

BACKGROUND

The eOracle target contracts consist of four smart contracts, the `EOFeedManager`, the `EOFeedVerifier`, the `EOFeedAdapter` and the `BLS` contract.

The `EOFeedManager` is responsible for receiving feed updates from whitelisted publishers, verifying them using `EOFeedVerifier`, and storing the verified data for access by other smart contracts.

The `EOFeedVerifier` handles the verification process of update payloads, ensuring the integrity and authenticity of the price feed updates. The update payload includes a Merkle root signed by eOracle validators and a Merkle path to the leaf containing the data. The verifier stores the current validator set in its storage and ensures that the Merkle root is signed by a subset of this validator set with sufficient voting power.

The `EOFeedAdapter` provides an `AggregatorV3` style interface to the feed data retrieval operations of the `EOFeedManager`.

The `BLS` contract provides the low-level cryptographic primitives which support the signature verification mechanisms used by the `EOFeedVerfier` contract.

SETTING & CAVEATS

This audit report mainly covers the delta of the changes of specific contracts included in PR #181 of the repository Eoracle/target-contracts of the eOracle protocol at commit be8459ad66f974e7e6a4a73744adb679825b5f04.

Two auditors worked on the codebase for 3 days on the following contracts:

```
src/
├── EOFeedManager.sol
├── EOFeedVerifier.sol
├── adapters/
│   └── EOFeedAdapter.sol
└── common/
    └── BLS.sol
```

As part of the audit, we also reviewed the fixes of the issues included in the report, which were delivered as new commits in PR #181 (up to commit 7a318d7, which is the merge commit of PR #213). We have found that they have been implemented correctly.

The audit's main target is security threats, i.e., what the community understanding would likely call "hacking", rather than the regular use of the protocol. Functional correctness (i.e. issues in "regular use") is a secondary consideration. Typically it can only be covered if we are provided with unambiguous (i.e. full-detail) specifications of what is the expected, correct behavior. In terms of functional correctness, we often trusted the code's calculations and interactions, in the absence of any other specification. Functional correctness relative to low-level calculations (including units, scaling and quantities returned from external protocols) is generally most effectively done through thorough testing rather than human auditing.

PROTOCOL-LEVEL CONSIDERATIONS

ID	Description	STATUS
P1	Required validations on the registered BLS validator keys	**ACKNOWLEDGED**

The protocol uses the BLS signature verification scheme to validate efficiently signed messages from multiple validators. In the previous version, the validator keys lived in the `G2` group and the signatures on `G1`. However, the checks required and the operations of `G2` in general are way more expensive to implement on-chain compared to the ones for the `G1` group. For that reason, this version was refactored to implement a new optimized method that allows both the validators' keys and the signature to live on `G1`. The method is described in this article (Optimized BLS multisignatures on EVM - Geometry).

However, as stated also there, there are some checks required to be performed to ensure specific properties for the keys and prevent known attacks from exploiting the validation process which could otherwise allow messages to be verified when they should not. More specifically:

- The Proof of Possession of the validators' keys should be performed to ensure that they really own the private key from which their public keys were generated.

- This optimized method requires the validators also to produce their public keys on `G2` using the same private key as the one used to generate their `G1` keys, which are stored on-chain. Thus, it is required to also ensure that both keys were generated by the same private key.

In the previous version, a similar concern was raised as the Proof of Possession check was missing from the Target Contracts of the protocol, which was still required for the proper operation of the BLS verification process. However, the team informed us that their Middleware component performs such checks outside the scope of these contracts. We mention these requirements here again so that it is ensured that the protocol sufficiently and properly handle all of them to ensure its integrity.

It should also be noted that the function that sets the validator keys on-chain is an `onlyOwner` function. This had raised some centralization concerns (see N1 in the previous report), which were being worked on but still applicable.

P2	Upgradeability considerations	**ACKNOWLEDGED**

The storage layouts of the `EOFeedAdapter`, `EOFeedManager` and `EOFeedVerifier` contracts were changed in the current version in a way that makes them incompatible with the previous versions. As a result, it is not safe to use them to update the existing proxies directly as the storage will end up broken and the data corrupted. We mention this here for visibility so that the new contracts are properly used to ensure the integrity of the data.

VULNERABILITIES & FUNCTIONAL ISSUES

This section details issues affecting the functionality of the contract. Dedaub generally categorizes issues according to the following severities, but may also take other considerations into account such as impact or difficulty in exploitation:

Category	Description
CRITICAL	Can be profitably exploited by any knowledgeable third-party attacker to drain a portion of the system's or users' funds OR the contract does not function as intended and severe loss of funds may result.
HIGH	Third-party attackers or faulty functionality may block the system or cause the system or users to lose funds. Important system invariants can be violated.
MEDIUM	Examples: • User or system funds can be lost when third-party systems misbehave. • DoS, under specific conditions. • Part of the functionality becomes unusable due to a programming error.
LOW	Examples: • Breaking important system invariants but without apparent consequences. • Buggy functionality for trusted users where a workaround exists. • Security issues which may manifest when the system evolves.

Issue resolution includes "dismissed" or "acknowledged" but no action taken, by the client, or "resolved", per the auditors.

CRITICAL SEVERITY:

[No critical severity issues]

HIGH SEVERITY:

[No high severity issues]

MEDIUM SEVERITY:

[No medium severity issues]

LOW SEVERITY:

ID	Description	STATUS
L1	Wrong mod operation in the delineation factor calculation	**RESOLVED**

***Resolution*:**
The value of the order of `G1` (`R`) was added in the `BLS` contract and was used to perform the mod operation on the delineation factor (`gamma`).

The verification of the signatures in the `BLS` contract was refactored to follow a more optimized implementation (Optimized BLS multisignatures on EVM - Geometry) that benefits from the simpler and cheaper operations on the `G1` curve.

However, the optimized method uses a delineation factor to separate the combined pairings to minimize the operations made. However, this factor is applied as the scalar and its value calculation should be made using the order of the `G1` group instead(i.e. `R`).

Since `N > R` in `BN254`, an adversary to exploit this issue should find a value of the delineation factor (`gamma`) equal to: $gamma\%N = k * r$ for any integer k. `gamma` is computed as the `keccak256` value of several prover controlled parameters which renders the exploit unfeasible as it is equal to finding a pre-image of the hash.

Nevertheless, this is still a misalignment from the correct implementation which can be easily fixed by performing the mod operation with the curve's `R` value.

L2	Inconsistency in the negation of the `G1` points	**RESOLVED**

***Resolution*:**
The implementation was updated to the suggested one and the operation was simplified by removing the extra checks.

In the `BLS` contract, the `neg` function was added to return the negated value of a point on `G1`. The negation of a `(x,y)` point results in the `(x,-y)` point. However, to negate `y` we need to subtract it from the fields order which is `N`. However, the current implementation is misconfigured and returns values for `y` in the range `[1,N]` instead of `[0,N-1]` as expected.

`BLS::neg:848`

```
function neg(uint256[2] memory p) public pure returns (uint256[2] memory)
{
    // The prime q in the base field F_q for G1
    if (p[0] == 0 && p[1] == 0) {
        return [uint256(0), uint256(0)];
    } else {
        // Dedaub:
        // It returns values in [1,N] instead of [0,N-1] for y.
        return [p[0], N - (p[1] % N)];
    }
}
```

The more accurate operation would be: $(N - p[1])\%N$ which also removes the need for special casing the function for the `(0,0)` point while also ensuring that the results are returned in the right range.

OTHER / ADVISORY ISSUES:

This section details issues that are not thought to directly affect the functionality of the project, but we recommend considering them.

ID	Description	STATUS
A1	Functional and role overlap in the `EOFeedManager` functions that set the supported feeds	**DISMISSED**

***Comments*:**
The EOracle team has indicated that the role overlap is intentional by design. While the owner has full configuration privileges, the feed deployer role is limited to feed deployment only, providing a more granular security model.

The `EOFeedManager` contract has an `onlyOwner setSupportedFeeds()` function which can set or reset the supported feeds, and an `onlyFeedDeployer addSupportedFeeds()` function which can set supported fields. These two functions overlap in functionality and role. For instance, while the owner can make a field unsupported, the feed deployer can make it supported again. We recommend consolidating these two functions into a single function and role to avoid redundant and overlapping responsibilities.

A2	Several `EOFeedAdapter` functions do not use the round parameter	**RESOLVED**

***Resolution*:**
Checks were added to the functions to only support the latest round IDs and revert on any other request preventing any misuse of the returned values.

The `getRoundData()`, `getAnswer()` and `getTimestamp()` functions have a round parameter, but they do not use it and instead return the values of the latest round. We understand that these functions are there to implement the `AggregatorV3Interface`, but we would like to point out that such an implementation of the interface is not the expected one. Perhaps it would make more sense to revert with an appropriate error message if the interface cannot be satisfied. Alternatively, it should be made very clear to consumers of `EOFeedAdapter` that the implementation is not the expected one.

A3	Minor optimization when registering the validator keys	**ACKNOWLEDGED**

***Comments*:**
The eOracle team has indicated that they will maintain the current implementation for code readability.

The `EOFeedVerifier` contract is used to verify the payloads to update the feeds. The previous implementation utilized validator public keys on the `G2` group of the `BN254` curve, but for efficiency the keys were transferred to `G1` which has much simpler and cheaper operations. Thus, the protocol aggregates all the validators' public keys on `G1` during their registration and the aggregated key is stored in storage to be used later for the signature verification. However, the iteration starts with the point `(0,0)` and then aggregates all the keys by adding them individually, but you could initialize the aggregated key with the 1st validator's public key and avoid 1 `ecadd` operation as a minor optimization.

A4	The `__gap` arrays were not reduced after adding new variables	**RESOLVED**

Comments:
The `__gap` array of the `EOFeedManager` contract was updated to reflect the changed storage layout.

In the `EOFeedManager` contract, 2 new variables were added occupying two more slots in storage. In `EOFeedVerifier`, though, 3 variables (3 slots) were removed and a fixed-size array was added occupying 2 slots. However, the `__gap` arrays were not updated to reflect these changes contrary to `EOFeedAdapter` which also got some new variables reducing the corresponding array.

DISCLAIMER

The audited contracts have been analyzed using automated techniques and extensive human inspection in accordance with state-of-the-art practices as of the date of this report. The audit makes no statements or warranties on the security of the code. On its own, it cannot be considered a sufficient assessment of the correctness of the contract. While we have conducted an analysis to the best of our ability, it is our recommendation for high-value contracts to commission several independent audits, a public bug bounty program, as well as continuous security auditing and monitoring through Dedaub Security Suite.

ABOUT DEDAUB

Dedaub offers significant security expertise combined with cutting-edge program analysis technology to secure some of the most prominent protocols in DeFi. The founders, as well as many of Dedaub's auditors, have a strong academic research background together with a real-world hacker mentality to secure code. Protocol blockchain developers hire us for our foundational analysis tools and deep expertise in

program analysis, reverse engineering, DeFi exploits, cryptography and financial mathematics.