EigenLayer Middleware

Smart Contract Security Assessment

Feb 05, 2024

ABSTRACT

Dedaub was commissioned to perform a security audit of the EigenLayer middleware functionality, as well as EigenDA, the first AVS (actively validated service) to launch on EigenLayer. The audit was over smart contract code and included a review of the cryptographic implementation as well as overall code logic. The code, documentation, and supporting elements (esp. test cases) are of excellent quality. However, given the generality of the infrastructure, much of the correctness burden is on the client of the middleware.

BACKGROUND

EigenLayer-middleware is a set of contracts that offer the necessary functionality for interacting with the EigenLayer base (restaking) infrastructure. This set of contracts is expected to serve the role of both static library and runtime support for AVSs. EigenDA, a data availability service, is the first AVS used to showcase and refine the middleware. For more detailed reading, there is extensive documentation in the respective repos as well as in the main EigenLayer document repository.

SETTING & CAVEATS

The audit report is over the contracts of repository https://github.com/Layr-Labs/eigenlayer-middleware, branch `m2-mainnet`, at commit `2856834e1f90133ed692cbc25f2880e3769c5c80` and of repository https://github.com/Layr-Labs/eigenda/, branch `m2-mainnet-contracts`, at commit `d06dec1a94c4e8082355f5e323d763e50d0712e4`.

2 auditors worked on the codebase for **15** days on the following contracts:

```
eigenlayer-middleware/src/
├── BLSApkRegistry.sol
├── BLSApkRegistryStorage.sol
├── BLSSignatureChecker.sol
├── IndexRegistry.sol
├── IndexRegistryStorage.sol
├── interfaces/
│   ├── IBLSApkRegistry.sol
│   ├── IBLSSignatureChecker.sol
│   ├── IDelayedService.sol
│   ├── IIndexRegistry.sol
│   ├── IRegistryCoordinator.sol
│   ├── IRegistry.sol
│   ├── IServiceManager.sol
│   ├── ISocketUpdater.sol
│   └── IStakeRegistry.sol
├── libraries/
│   ├── BitmapUtils.sol
│   └── BN254.sol
├── OperatorStateRetriever.sol
├── RegistryCoordinator.sol
├── RegistryCoordinatorStorage.sol
├── ServiceManagerBase.sol
├── StakeRegistry.sol
└── StakeRegistryStorage.sol

eigenda/contracts/src/
├── core/
│   ├── EigenDAServiceManager.sol
│   └── EigenDAServiceManagerStorage.sol
├── Imports.sol
├── interfaces/
│   └── IEigenDAServiceManager.sol
├── libraries/
│   ├── EigenDAHasher.sol
│   └── EigenDARollupUtils.sol
└── rollup/
    └── MockRollup.sol
```

There are some caveats that should be spelled out, i.e., threats that are out-of-scope for this report. These provide an important context for the rest of the report.

A major caveat in the case of EigenDA is that the smart contract code constitutes a small part of the overall implementation. There are about 400 lines (excluding interfaces) of smart contract code and many thousands of lines of Go (i.e., off-chain) code. It is hard to separate the correctness obligations of the off-chain code relative to the smart contract code. Therefore, most of the correctness is left to off-chain code. Even checks defined inside smart contracts (such as the function `EigenDARollupUtils::verifyBlob`) are intended fully to be applied off-chain (the function is not called by any other smart contract code) and sanity-check inputs against other inputs, not against on-chain state.

Generally, the line between on-chain and off-chain validation in EigenDA is hard to discern and it seems possible that it may shift in the future (e.g., with more decentralization). For instance, issue A1 results directly from the difficulty of telling why some validation is done on-chain whereas other is performed off-chain. In principle, given that the current EigenDA architecture is centralized and the obligation of checking is entirely left to the batch checker, all validation (including confirmation of batches/ signatures) could be performed off-chain, with only final records committed on-chain.

As a result, we assume a well-functioning off-chain facility that maintains the documented properties of EigenDA (including tolerance to reorgs, correct economic incentives, etc.) and calls the smart contracts with correct inputs.

Another caveat concerns the guarantees provided by the base EigenLayer protocol. Straightforward attacks, such as temporarily achieving a majority stake only to influence a quorum, are economic threats in the heart of EigenLayer. Clearly, if the base protocol is not sufficiently protected against a malicious large-stake owner then EigenLayer middleware is also unsafe: the middleware trusts the main EigenLayer delegation manager contract and assigns validation power based on the stakes it reports. We do not report any such threats as issues of EigenLayer middleware, although clearly any qualifications of the base EigenLayer protocol economic security have to be considered in the context of every middleware client, i.e., every AVS deployed on EigenLayer. (For instance, it is perfectly conceivable that some AVS will not be able to tolerate instantaneous loss of control to a malicious party that temporarily holds a high stake, although it can accept the risk if the stake is held for a time long enough for misbehavior to incur economic penalties.)

Finally, the audit carefully reviewed cryptographic assumptions and the implementation. The assumptions appear entirely safe, but remain assumptions. This trivially includes widely-accepted cryptographic assumptions, but also, more meaningfully, simple complexity calculations (e.g., truncating public key hashes to 24 bytes appears safe to us as it did to the developers), as well as protocol design decisions. Specifically, the main optimization, at the heart of the implementation, is based on an informal article: https://geometry.xyz/notebook/Optimized-BLS-multisignatures-on-EVM. We reviewed in depth the article's cryptography, proof-of-concept code, and final implementation of the mechanism. Although we are convinced of correctness, the article does not supply rigorous proofs.

PROTOCOL-LEVEL CONSIDERATIONS

As a middleware facility, EigenLayer-middleware is to be used in a large variety of settings. This makes some protocol-level considerations rise in importance. We itemize issues that we think are vectors of general threats. These are warnings for all future clients of EigenLayer-middleware (beyond the current EigenDA, which is unaffected).

ID	Description	STATUS
P1	Race conditions and records inconsistent with public structures	**LARGELY RESOLVED (6e010fee)**

Resolution:

The item is, for all practical intents, resolved: the `checkSignatures` function, which is the foremost entry point, no longer allows processing based on current-block data. The underlying potential for races remains, in the sense that all data structures return values for the current block number, although these may be overwritten. Thus, view functions can return data for the current block and these data can trigger actions that will fail. However, mere awareness of this potential in clients should be enough.

Although `BLSSignatureChecker::checkSignatures` checks that the reference block number is not in the future, the **current** block (i.e., the next to be produced) is accepted as reference:

```
function checkSignatures(
      bytes32 msgHash,
      bytes calldata quorumNumbers,
      uint32 referenceBlockNumber,
      NonSignerStakesAndSignature memory params
){
      ...
      require(referenceBlockNumber <= uint32(block.number),
"BLSSignatureChecker.checkSignatures: invalid reference block");
```

As a consequence, users of eigenlayer-middleware could be tempted to submit confirmations for the current block, in an attempt to provide them as early as possible. This practice, however, is risky, as it creates race conditions between the signature check and other updates that might be performed in the same block.

If an update happens in the same block **before** `checkSignatures`, it might violate some of the data included in the check, such as the APK or the stake history indexes, causing the signature check to fail. In issues M1 and L2 we discuss in detail two scenarios in which such a race condition could be caused maliciously to achieve a DoS attack.

Similarly, an update might happen in the same block but **after** checkSignatures. In this case, the signature check will succeed, but there will be an inconsistency between the data being checked and the information stored for that block, which could have undesirable consequences for the protocol. For instance, the signatoryRecord hashed and stored by EigenDAServiceManager::confirmBatch could be referring to a block with data inconsistent with the globally-maintained data about that block. To reconstruct records, one would need to replay past transactions.

As a consequence we recommend strengthening the above require to ensure that only blocks strictly in the past can be used as reference. Alternatively, AVS implementations should be careful to ensure they can tolerate such races and apparent-inconsistency in records.

P2	msgHash parameter of checkSignatures should be non-tainted in clients of EigenLayer middleware	**RESOLVED, documentation (bfe962b0)**

Clients of EigenLayer-middleware will call its main validation function, checkSignatures:

```
function checkSignatures(
      bytes32 msgHash,
      bytes calldata quorumNumbers,
      uint32 referenceBlockNumber,
      NonSignerStakesAndSignature memory params
)
```

Clients should be careful to ensure that argument msgHash is indeed a hash (i.e., collision-resistant), and not directly controlled by an untrusted party. If the latter were to happen, multiple purported signed messages (likely garbage) could be validated.

The reason is that the argument is being passed to function trySignatureAndApkVerification and eventually to hashToG1, which (despite its name) is **not** a collision-resistant hash function: it directly applies a modulo operation to its argument, before attempting to find a group element.

BN254::hashToG1:293

```
function hashToG1(bytes32 _x) internal view returns (G1Point memory) {
  uint256 beta = 0;
  uint256 y = 0;
  uint256 x = uint256(_x) % FP_MODULUS;
  …
```

P3	Multiple valid sequences of signers possible	INFO

Similarly to item P2, but even more obviously, clients should be aware that the sets of signers/non-signers/signed messages that can be correctly validated will not be unique. E.g., if a set of non-signers NS would validate correctly, the set can be augmented by one more purported non-signer, by mere subtraction from the signatures and public keys sets. If the larger non-signer set (i.e., smaller signer set) still satisfies stake constraints, validation will pass. This is an "obvious" concern and is unlikely to pose problems for most protocols, but we note it here for completeness, for consideration in future clients (especially highly-decentralized ones, with multiple independent and potentially racing callers of checkSignatures).

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:

ID	Description	STATUS
M1	StakeUpdate-based DoS on `confirmBatch` (or any future client of `checkSignatures`) if `referenceBlockNumber` is the current block	**RESOLVED (6e010fee)**

As discussed in P1, `BLSSignatureChecker::checkSignatures` allows the use of the current block as the `referenceBlockNumber`. As a consequence, EigenDA could be configured to call `EigenDAServiceManager::confirmBatch` with the **current block** (i.e., the next to be produced) as reference, with the possible goal of providing confirmations as early as possible. Such a practice, however, could allow an **external** malicious entity (not necessarily an operator) to cause the `confirmBatch` transaction to fail. We describe the concrete attack below.

In order to verify the operators' stakes at the reference block, `checkSignatures` requires the caller to provide indexes to the stake history list (so that the list does not need to be traversed), for instance `params.totalStakeIndices` should point to the total stake record for each quorum. These indexes are validated by `StakeRegistry::_validateOperatorStakeUpdateAtBlockNumber`, by checking that the corresponding `StakeUpdate` happened before or at the referenced block, and that the next update happened later (or that the update is the last in the list).

```
function _validateOperatorStakeUpdateAtBlockNumber(
      StakeUpdate memory operatorStakeUpdate,
      uint32 blockNumber
) internal pure {
      /**
      * Validate that the update is valid for the given blockNumber:
      * - blockNumber should be >= the update block number
      * - the next update block number should be either 0 or strictly
greater than blockNumber
      */
      require(
       blockNumber >= operatorStakeUpdate.updateBlockNumber,
       "StakeRegistry._validateOperatorStakeAtBlockNumber:
operatorStakeUpdate is from after blockNumber"
      );
      require(
       operatorStakeUpdate.nextUpdateBlockNumber == 0 ||
       blockNumber < operatorStakeUpdate.nextUpdateBlockNumber,
       "StakeRegistry._validateOperatorStakeAtBlockNumber: there is a
newer operatorStakeUpdate available before blockNumber"
      );
}
```

Now in order to call `confirmBatch` for the current block, the batch confirmer needs to provide an index to the **last** history entry in the list (that is an entry having `nextUpdateBlockNumber == 0`). That `StakeUpdate` happened in a **previous** block (the batch confirmer is an EOA so cannot reliably perform an update and call `confirmBash` in the same block), but it is still valid for the current block since it is the last in the list.

The DoS possibility then is as follows: upon observing a pending transaction containing `confirmBatch` for the next block, any malicious entity could front-run it with a call to `RegistryCoordinator::updateOperators`. Such an update essentially invalidates the index included in the call to `confirmBatch`. The entry's `nextUpdateBlockNumber` will become equal to the current block number, causing `_validateOperatorStakeUpdateAtBlockNumber` to fail (the newly created history entry should be used instead of the old one).

To prevent such DoS opportunities, `confirmBatch` can always be called for past blocks, which could also be enforced on-chain by modifying the `referenceBlockNumber` check in `BLSSignatureChecker::checkSignatures.`

LOW SEVERITY:

ID	Description	STATUS
L1	Idiom results in nested loop, can be optimized	**RESOLVED (b51fda48)**

The code idiom, below, using `bytes.concat`, incurs a loop, creating new copies of the intermediate array. This results in O(n^2) complexity, which is unnecessary and can be optimized with pre-allocation of the final array.
[The issue was discussed and addressed during the audit period.]
`BitmapUtils::bitmapToBytesArray:126`

```
for (uint256 i = 0; i < 256; ++i) {
  // construct a single-bit mask for the i-th bit
  bitMask = uint256(1 << i);
  // check if the i-th bit is flipped in the bitmap
  if (bitmap & bitMask != 0) {
      // if the i-th bit is flipped, then add a byte … to the `bytesArray`
      bytesArray = bytes.concat(bytesArray, bytes1(uint8(i)));
  }
}
```

L2	APK-based DoS on `confirmBatch` (or any future client of `checkSignatures`) if `referenceBlockNumber` is the current block	**RESOLVED (6e010fee)**

In P1 we discussed that allowing the current block to be used as a reference in `BLSSignatureChecker::checkSignatures` can be potentially problematic. In M1 we described in detail how it can lead to DoS on `confirmBatch` by front-running `confirmBatch` with a stake update. Here, we briefly describe a similar DoS possibility, this time via an APK update.

Consider a pending transaction with a call to `confirmBatch` for the current block. Such a transaction contains a signature built w.r.t. to the aggregate public key registered for that block. If this APK is modified before executing `confirmBatch`, it will cause the transaction to fail for two reasons:

- The APK will not match the one of the signature
- The APK index (in `params.quorumApkIndices`) will no longer match the reference block.

An APK update can be performed:

- Either by a malicious registered operator (independently from whether he has signed the batch or not), who could deregister and re-register afterwards.
- Or by a malicious unregistered operator (if the conditions for a new registration are met).

Although this attack is harder to execute than M1, it further reinforces our view that only past blocks should be used as reference.

CENTRALIZATION ISSUES:

ID	Description	STATUS
N1	Permissioned accounts and centralized services	**INFO**
The EigenLayer-middleware contracts include clearly-identified permissioned roles, such as the owner of the RegistryCoordinator. Furthermore, the EigenDA protocol operates currently in a centralized manner: a single permissioned account can confirm batches. This mitigates some potential issues (e.g., P3, A1) but creates a trust obligation.		

OTHER / ADVISORY ISSUES:

This section details issues that are not thought to directly affect the functionality of the project, but we recommend considering them. These issues are explicitly labeled "Info" and not "Open". Before acting on such issues, developers should also confirm them to the best of their ability.

ID	Description	STATUS
A1	`EigenDAServiceManager::confirmBatch` does not fully validate its inputs	**INFO**
It is not clear what are the correctness obligations of the `confirmBatch` function of EigenDAServiceManager. It is reasonable to assume that the input has been validated by the permissioned caller, the batch confirmer. However, some input validation checks (e.g., on the `referenceBlockNumber`) are performed on-chain, whereas others are not. Notably, the length of the `batchHeader.quorumNumbers` and that of		

batchHeader.quorumThresholdPercentages are not checked to match. The latter can well be zero, eschewing all stake checks.

A2	The adversaryThresholdPercentage field of IEigenDAServiceManager::QuorumBlobParam seems unused.	**INFO**

The adversaryThresholdPercentage field is only checked against others that are also supplied externally, but is not otherwise used in smart contract code.

A3	Inaccurate revert message, zero apkHash consideration	**INFO**

In BLSApkRegistry::getApkIndicesAtBlockNumber the message below is not entirely accurate:

BLSApkRegistry::getApkIndicesAtBlockNumber:212

```
if (quorumApkUpdatesLength == 0 ||
   blockNumber < apkHistory[quorumNumber][0].updateBlockNumber) {
     revert("BLSApkRegistry.getApkIndicesAtBlockNumber: blockNumber is
before the first update");
   }
```

Strictly speaking, the first checked condition indicates that the block is before quorum initialization, not before the first update.

It is possible that the revert message is the desired behavior and that the current behavior is a bug. I.e., that if there has been no quorum update (with registering operators) there should be no apkHash (of zero) returnable. We considered whether returning a zero apkHash is a threat and could not devise an attack.

A4	Loops always iterate upwards	**INFO**

There is a general aversion in the codebase to loops that do not iterate upwards. This is probably a matter of taste, but we just bring up that the code could be arguably more direct with a downwards loop. E.g., instead of
RegistryCoordinator::_getQuorumBitmapIndexAtBlockNumber:764

```
for (uint256 i = 0; i < length; i++) {
  index = uint32(length - i - 1);
  if (_operatorBitmapHistory[operatorId][index].updateBlockNumber <=
      blockNumber) {
    return index;
  }
}
```

to have:

```
for (uint32 i = uint32(length); i > 0; i--) {
  if (_operatorBitmapHistory[operatorId][i - 1].updateBlockNumber <=
      blockNumber) {
    return i - 1;
  }
}
```

Other instances of the upward-loop-that-is-really-downward pattern occur in BLSApkRegistry::getApkIndicesAtBlockNumber, IndexRegistry::_operatorCountAtBlockNumber, IndexRegistry::_operatorIdForIndexAtBlockNumber, StakeRegistry::_getStakeUpdateIndexForOperatorAtBlockNumber.

A5	Misleading function name	**INFO**

The name of the function (and its argument) below is slightly misleading:
StakeRegistry::_validateOperatorStakeUpdateAtBlockNumber:445

```
function _validateOperatorStakeUpdateAtBlockNumber(
      StakeUpdate memory operatorStakeUpdate,
      uint32 blockNumber
) internal pure { … }
```

The function is used to validate StakeUpdate entries not just of the `operatorStakeHistory` mapping but also of the `totalStakeHistory` mapping.

A6	Minor optimization	INFO

The function below admits minor optimization. Notably, it is on the control-flow path of `CheckSignatures`, so this is probably worth applying.

`BitmapUtils::orderedBytesArrayToBitmap:62`

```
function orderedBytesArrayToBitmap(bytes memory orderedBytesArray,
                     uint8 bitUpperBound) internal pure returns (uint256) {
 uint256 bitmap = orderedBytesArrayToBitmap(orderedBytesArray);

 if (bitmap != 0) {
  require(
   uint8(orderedBytesArray[orderedBytesArray.length - 1]) < bitUpperBound,
   "BitmapUtils.orderedBytesArrayToBitmap: bitmap exceeds max value"
  );
 }
 return bitmap;
}
```

The check could be replaced by "`(1 << bitUpperBound) > bitmap`" to eliminate the need for memory access.

A7	Misleading comment	INFO

The comment below is not accurate.

`IBLSSignatureChecker:19`

```
struct NonSignerStakesAndSignature {
  …
  BN254.G2Point apkG2;
  // is the aggregate G2 pubkey of all signers and non signers
  …
}
```

The supplied group-2 pubkey is only for the signers.

A8	Invalid comments	INFO

The comments below are copy-paste or leftover (old name?) errors.

BitmapUtils::isArrayStrictlyAscendingOrdered:100

```
// loop through each byte in the array to construct the bitmap
for (uint256 i = 1; i < bytesArray.length; ++i) {
```

StakeRegistryStorage:19

```
/// @notice Maximum length of dynamic arrays in the
`strategiesConsideredAndMultipliers` mapping.
uint8 public constant MAX_WEIGHING_FUNCTION_LENGTH = 32;
```

A9	Grammar or spelling errors in comments	INFO

There are a few spelling errors in comments:

StakeRegistry:253

```
 * @notice Modifys the weights of existing strategies for …
```

StakeRegistry:394

```
 * @dev This function …. This is a concious choice,
```

EigenDAServiceManager:62

```
 * - submitting data availabilty certificates,
```

And some grammar errors:

IndexResistry::_operatorIdForIndexAtBlockNumber:292

```
// we should only it this if the operatorIndex was never used before …
```

ServiceManagerBase:16

```
* This contract can inherited from or simply used as a point-of-reference.
```

IBLSApkRegistry:9

```
* @title Minimal interface for a registry that …for among many quorums.
```

IBLSSignatureChecker:26

```
uint32[] totalStakeIndices; // is the indices of each quorums total stake
```

IEigenDAServiceManager:50

```
// the must have signed …
```

A10	Future compatibility threat	INFO

Some Ethereum Improvement Proposals suggest that the check "`tx.origin == msg.sender`" may become ineffective in the future, for ensuring that the `msg.sender` is an EOA. The possibility currently seems remote and the potential impact to the protocol is small.

EigenDAServiceManager::confirmBatch:71

```
function confirmBatch(
   BatchHeader calldata batchHeader,
   NonSignerStakesAndSignature memory nonSignerStakesAndSignature
) external onlyWhenNotPaused(PAUSED_CONFIRM_BATCH) onlyBatchConfirmer() {
   // make sure the information needed to derive the non-signers and
   // batch is in calldata to avoid emitting events
   require(tx.origin == msg.sender,
    "EigenDAServiceManager.confirmBatch: header and nonsigner data must be in calldata");
```

A11	Unnecessary code	INFO

There are some unnecessary checks , which currently never fail.

RegistryCoordinator::updateOperatorsForQuorum:291

```
uint192 quorumBitmap = uint192(
 BitmapUtils.orderedBytesArrayToBitmap(quorumNumbers, quorumCount));
require(
 _quorumsAllExist(quorumBitmap),
  "RegistryCoordinator.updateOperatorsForQuorum: some quorums do not exist"
);
```

(The `quorumCount` argument in the first statement ensures that the quorum numbers are all below this bound, hence they exist, so the `require` cannot fail.)

`StakeRegistry::_getStakeUpdateIndexForOperatorAtBlockNumber:280`

```
function _getStakeUpdateIndexForOperatorAtBlockNumber(
      bytes32 operatorId,
      uint8 quorumNumber,
      uint32 blockNumber
) internal view returns (uint32) {
  uint256 length = operatorStakeHistory[operatorId][quorumNumber].length;
  for (uint256 i = 0; i < length; i++) {
      if (operatorStakeHistory[operatorId][quorumNumber]
                      [length - i - 1].updateBlockNumber <= blockNumber) {
          uint32 nextUpdateBlockNumber =
             operatorStakeHistory[operatorId][quorumNumber]
                      [length - i - 1].nextUpdateBlockNumber;
          require(nextUpdateBlockNumber == 0 ||
                  nextUpdateBlockNumber > blockNumber,
             "StakeRegistry._getStakeUpdateIndexForOperatorAtBlockNumber:
operatorId has no stake update at blockNumber");
          return uint32(length - i - 1);
      }
  }
  revert("StakeRegistry._getStakeUpdateIndexForOperatorAtBlockNumber: no
stake update found for operatorId and quorumNumber at block number");
}
```

The `require` seems unnecessary: if there is a record in `operatorStakeHistory`, the last record's `nextUpdateBlockNumber` will always be zero.

Incidentally, the above function can be minorly gas-optimized, by storing memory and storage pointers instead of calculating the same addresses repeatedly. We do not include this and other such items in the report because the function appears to be called only off-chain.

A12	Unused member	INFO

BN254:80

```
bytes32 internal constant powersOfTauMerkleRoot = // Dedaub: unused
      0x22c998e49752bbb1918ba87d6d59dd0e83620a311ba91dd4b2cc84990b31b56f;
```

A13	Compiler bugs	INFO

The code is compiled with Solidity 0.8.12. This version has some known bugs, which we do not believe to affect the correctness of the contracts.

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 Watchdog.

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.