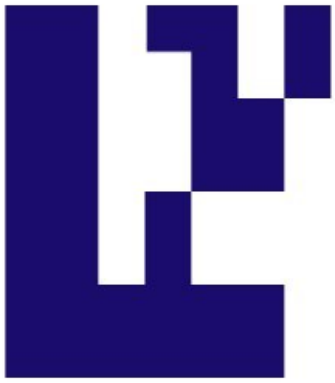
Eigenlayer Middleware

Smart Contract Security Assessment

April 30, 2025



ABSTRACT

Dedaub was commissioned to perform a security audit of the Eigenlayer middleware contracts that provide a higher-level interface to the core protocol.

BACKGROUND

Eigenlayer is a protocol that allows stakers to restake assets (native ETH or any ERC20 token) across services. Stakers restake their assets by delegating them to an operator. In turn, operators register to provide off-chain services for Autonomous Verifiable Services (AVS), providing evidence of their execution on-chain. For this, AVSs distribute tokens as rewards for the stakers and operators.

The Eigenlayer protocol is implemented in a set of core contracts, while providing a set of middleware contracts intended for the AVSs to use as a higher-level interface for the core protocol. These middleware contracts are not necessary for interaction with the protocol, although their interfaces are. They are recommended for use by Eigenlayer, and as such their security is important to the security of the protocol.

The Eigenlayer protocol previously only supported de/registration of operators and rewards. This audit covers the introduction of the following new features:

- **Operator Sets**: groupings of operators per AVS, meant to organise them depending on the service/task they provide and rewards to be received from performing the service. This is implemented in the core contracts (in AllocationManager.sol).

  A related grouping in the middleware is that of quorums, in fact every quorum has an associated operator set. Quorums put further configuration details on top of operator sets (maximum number of operators allowed, the minimum amount of stake of each operator, and the tokens allowed to be restaked and their ‘voting’ value).

- **Slashing**: Misbehaving operators can now be slashed by AVSs from operator sets, with their slashable stake burned. This is implemented in the core contracts (AllocationManager.sol), but triggered by the AVS. The middleware provides two different modalities for slashing: instant and vetoable (giving a veto committee a certain amount of time to veto the slashing).

- **Permission delegation**: EOAs and contracts can now specify other EOAs and contracts as their appointees, allowing them to call certain functions in their stead. This is implemented in the core protocol contracts (PermissionController.sol), and used extensively to allow middleware contracts to interact with the core contracts (through AllocationManager.sol.)

Furthermore, the middleware provides support for pre-existing AVSs and their operators to migrate to operator sets, through the RegistryCoordinator contract. This allows pre-existing operators to run the old M2 quorum protocol (not under audit here) and the new one in tandem, and to eventually fully migrate.

SETTING & CAVEATS

This audit report mainly covers the contracts of the Github repository at commit `c7b1774176e105e966f9619572ec3feba8b7de31`, excluding contracts in src/unaudited. Furthermore, this audit includes two contracts from the Github repository at commit `493c773ce9917f996ef7bc56eb9c97a80134d53e`, namely src/contracts/core/AllocationManager.src and src/contracts/permissions/PermissionController.sol

Three auditors worked on the codebase for 2 weeks on the following contracts:

```
eigenlayer-middleware/src/
  ├── BLSApkRegistry.sol
  ├── BLSApkRegistryStorage.sol
```

```
├── BLSSignatureChecker.sol
├── BLSSignatureCheckerStorage.sol
├── EjectionManager.sol
├── EjectionManagerStorage.sol
├── IndexRegistry.sol
├── IndexRegistryStorage.sol
├── interfaces/
│   ├── IBLSApkRegistry.sol
│   ├── IBLSSignatureChecker.sol
│   ├── IECDSAStakeRegistry.sol
│   ├── IEjectionManager.sol
│   ├── IIndexRegistry.sol
│   ├── IInstantSlasher.sol
│   ├── IRegistryCoordinator.sol
│   ├── IServiceManager.sol
│   ├── IServiceManagerUI.sol
│   ├── ISlasher.sol
│   ├── ISlashingRegistryCoordinator.sol
│   ├── ISocketRegistry.sol
│   ├── IStakeRegistry.sol
│   └── IVetoableSlasher.sol
├── libraries/
│   ├── BitmapUtils.sol
│   ├── BN254.sol
│   ├── LibMergeSort.sol
│   ├── QuorumBitmapHistoryLib.sol
│   └── SignatureCheckerLib.sol
├── OperatorStateRetriever.sol
├── RegistryCoordinator.sol
├── RegistryCoordinatorStorage.sol
├── ServiceManagerBase.sol
├── ServiceManagerBaseStorage.sol
├── ServiceManagerRouter.sol
├── slashers/
│   ├── base/
│   │   ├── SlasherBase.sol
│   │   └── SlasherStorage.sol
│   ├── InstantSlasher.sol
│   └── VetoableSlasher.sol
├── SlashingRegistryCoordinator.sol
```

```
  ├── SlashingRegistryCoordinatorStorage.sol
  ├── SocketRegistry.sol
  ├── SocketRegistryStorage.sol
  ├── StakeRegistry.sol
  └── StakeRegistryStorage.sol

eigenlayer-contracts/src/
  └── contracts/
      ├── core/
      │   └── AllocationManager.sol
      └── permissions/
          └── PermissionController.sol
```

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	Optimization to the registration with churn process	**INFO**

For registration with churn, operators call `AllocationManager::registerForOperatorSets()` in the core contracts to register while specifying the churning information in the parameter `RegisterParams.data`.
This function calls `SlashingRegistryCoordinator::registerOperator()` in the middleware which performs the registration of the operator in the middleware. This function will first register the operator normally as if no churning is happening. Later it will check if a churn is needed by checking the new number of operators in the quorum and if it exceeds the max allowed number, it will perform the churning using `_kickOperator()`.
Interestingly, `_kickOperator()` performs a call back to the `AllocationManager::deregisterFromOperatorSets()` to deregister the kicked-out operator from the core contract. The latter function calls again the `SlashingRegistryCoordinator` contract in the middleware to perform a deregistration of the kicked-out operator from the middleware.
During this process, the registration of the new operator and deregistration of the kicked-out operator both involve performing the registration/deregistration in the sub-contracts of the middleware, namely `StakeRegistry` , `BLSApkRegistry` , and `IndexRegistry`. In particular, this involves:

- For `BLSApkRegistry` contract, the registration (deregistration) is adding (subtracting) the registered operator's public key to the `apkHistory` storage variable.
- For `StakeRegistry` contract, the registration (deregistration) updates the stake of the registered (deregistered) operator in the storage variable `operatorStakeHistory` and adds (subtracts) it to the total stake in `_totalStakeHistory` storage variable.
- For `IndexRegistry` contract, the registration increases the operators count in `_operatorCountHistory` and adds the operator id (hash of its public key) to the `_updateOperatorIndexHistory` storage variable. On the other hand, the deregistration decreases the operators count in `_operatorCountHistory` and

pops out the last added operator id from `_updateOperatorIndexHistory` by writing a dummy id (i.e., `OPERATOR_DOES_NOT_EXIST_ID`) to `_updateOperatorIndexHistory` and then replaces the deregistered operator id with new operator id in `_updateOperatorIndexHistory`.

One can see that this procedure of churning (i.e., registering and then deregistering) can be optimized by saving a lot of storage operations if the churning was done in a unique optimized execution path. In more detail:

- For `BLSApkRegistry` contract, the churning adds to `apkHistory` the difference between the registered and kicked out operator pk.
- For `StakeRegistry` contract, the churning updates the stake of the registered and kicked out operator in `operatorStakeHistory` (*no optimization here*) and then updates `_totalStakeHistory` by the diff of the stake.
- For `IndexRegistry` contract, the churning replaces the kicked-out operator id with new operator id immediately by pushing the new operator id to the index of the deregistered operator in `_updateOperatorIndexHistory.`

Such optimization is estimated to cut the number of storage operations approximately by half.

P2	Important security consideration for the off-chain AVS tasks	**INFO**

The function `checkSignatures()` accepts the hash of the msg (`msgHash`) in its input which represents the hash of the committed result of a specific AVS task by the operators. It is important to ensure that the message produced by an off-chain AVS task is not controlled by the adversary. If this guarantee fails, a malicious operator who can control this message can choose it to be equal to the registration message as `msg = "\x19\x01 || domainSeparator || keccak256(abi.encode(PUBKEY_REGISTRATION_TYPEHASH, operatorAddr))"`

where `operatorAddr` is the address of the malicious adversary and `domainSeparator` is defined in EIP-712. By setting `msg` to this value, all targeted operators who are contributing to the signature of this AVS task will hash and sign the message producing a valid aggregate signature `sigma_agg` of the registration message of the malicious operator. Consequently, the malicious aggregator can use this signature to register to all quorums with a rogue public key and compromise all the quorums where the targeted operators are part of. The attack goes as the following:

1. The attacker generates a secret key `sk1'` and public keys `pk1'` and `pk2'` then computes the public keys `pk1 = pk1' - sum(pk1_i)` and `pk2 = pk2' - sum(pk2_i)` where `sum(pk1_i)` and `sum(pk2_i)` are the sum of the public keys of the targeted operators.
2. The attacker signs `msg` with `sk1'` to calculate the signature `sigma'`. He then computes the signature `sigma = sigma' - sigma_agg`.
3. Finally, the attacker registers as an operator using the public keys `pk1` and `pk2` and the signature `sigma`.

The registration passes because it represents a valid signature of the registration message even though the attack does not actually know the secret key corresponding to public keys `pk1` and `pk2`.

From now on, the attacker can sign arbitrary messages on behalf of all the targeted operators without the targeted operators actually contributing to the signature. This applies for all quorums that the targeted operators are part of.

To ensure such attack is prevented we suggest the following recommendations:

1. Make sure that each AVS task has a unique output by appending a special domain specific string to the output of the task.
2. Warn the operators that they **MUST** use different public keys when registering to different AVSs. Failure to do so enables the attacker to create his own AVS which gives him control of the output of the AVS task thus controlling what the operators sign.

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	Operators can register to a full quorum without churning out any operator in `SlashingRegistryCoordinator`	**RESOLVED**

Resolution:
The function `SlashingRegistryCoordinator::_validateChurn()` is updated to require that the `operatorToKick` is registered for the quorum in this PR.

When an operator registers with churn to a full quorum using `SlashingRegistryCoordinator::registerOperator()`, the function `SlashingRegistryCoordinator::_kickOperator()` is called to kick out an operator from the quorum and free a place for the new registering operator. The later function checks if the kicked out operator is currently part of the concerned quorum by applying the following check:

```
function _kickOperator(address operator, bytes memory quorumNumbers)
internal virtual {
      OperatorInfo storage operatorInfo = _operatorInfo[operator];
      // Only proceed if operator is currently registered
```

```
        require(operatorInfo.status == OperatorStatus.REGISTERED,
OperatorNotRegistered());

        bytes32 operatorId = operatorInfo.operatorId;
        uint192 quorumsToRemove =
        uint192(BitmapUtils.orderedBytesArrayToBitmap(quorumNumbers,
quorumCount));
        uint192 currentBitmap = _currentOperatorBitmap(operatorId);

        emit kickingOperator(operator, currentBitmap, quorumsToRemove);
        // Check if operator is registered for all quorums we're trying to
remove them from
        // @Dedaub: This check allows the deregistration to be skipped
        if (quorumsToRemove.isSubsetOf(currentBitmap)) {
            _forceDeregisterOperator(operator, quorumNumbers);
        }
}
```

However, when the check fails, the transaction does not revert. Instead it just ignores the deregistration of the kicked out operator. So, if an operator registers with churn using a kicked operator that is not part of the registering quorum, then the operator will be registered to the full quorum without de-registering any of its already existing operators leading to exceeding the size limit of the max number of operators for that quorum and thus breaking an invariant.

To exploit this vulnerability, an operator willing to register to quorum A can choose the kicked out operator as an operator who is registered to any quorum but the quorum A. This will ensure that the kicked out operator has status `REGISTERED` and the check above fails.

Note that removing this check means `_forceDeregisterOperator()` is always called, which will in turn eventually call `AllocationManager::deregisterFromOperatorSets()`. The latter checks that the

kicked out operator is a member of the operator set, and reverts if not. Additionally, this requirement is enforced again in `SlashingRegistryCoordinator::_deregisterOperator()` after the allocation manager calls `SlashingRegistryCoordinator::deregisterOperator()`.
A proof of concept exploit is provided in the Appendix Proof of Concept - M1.

M2	Operators can register to a full quorum without churning out any operator in `RegistryCoordinator`	**RESOLVED**

Resolution:
The function `SlashingRegistryCoordinator::_validateChurn()` is updated to require that the `operatorToKick` is registered for the quorum in this PR.

The `RegistryCoordinator` contract inherits `StakeRegistryCoordinator` and is used for legacy support for old AVSs. The function `_kickOperator()` is overridden and the check for the `quorumsToRemove` being a subset of `currentBitmap` is removed which fixes the issue described in M1. However, another bug was introduced in the overridden implementation by removing the requirement for the deregistered operator to have `REGISTERED` status.

```
function _kickOperator(address operator, bytes memory quorumNumbers)
internal virtual override {
      OperatorInfo storage operatorInfo = _operatorInfo[operator];
      uint192 quorumsToRemove =
uint192(BitmapUtils.orderedBytesArrayToBitmap(quorumNumbers,
quorumCount));
      // @Dedaub: if the kicked out operator is not registered, the
deregistration is skipped
      if (operatorInfo.status == OperatorStatus.REGISTERED &&
!quorumsToRemove.isEmpty()) {
      // Allocate memory once outside the loop
      bytes memory singleQuorumNumber = new bytes(1);
      // For each quorum number, check if it's an M2 quorum
```

```
        for (uint256 i = 0; i < quorumNumbers.length; i++) {
                singleQuorumNumber[0] = quorumNumbers[i];

                if (isM2Quorum(uint8(quorumNumbers[i]))) {
                // For M2 quorums, use _deregisterOperator
                _deregisterOperator({operator: operator, quorumNumbers:
singleQuorumNumber});
                } else {
                // For non-M2 quorums, use _forceDeregisterOperator
                _forceDeregisterOperator(operator, singleQuorumNumber);
                }
        }
        }
}
```

So, for AVSs using the legacy support contract `RegistryCoordinator`, a similar exploit to the one in M1 can be performed by just using a random (non-existing) address for the kicked out operator.

A proof of concept exploit is provided in the Appendix Proof of Concept - M2.

LOW SEVERITY:

ID	Description	STATUS
L1	Wrong calculation of the total ejectable stakes	**RESOLVED**

Resolution:

A patch is applied to recalculate the `amountEjectableForQuorum` for each operator ejected in this PR.

The function `EjectionManager::amountEjectableForQuorum()` calculates the remaining amount of stakes an ejector can still eject at point in time. However, the calculation overestimates the value since it does not take into account the fact that the total stakes decrease after ejecting each operator. In particular, the function calculates the value based on all existing stakes before ejecting any operator. Consequently, this may allow an ejector to eject operators with total stakes exceeding the rate limit.
For example, given 10 operators each with 1 stake and `ejectableStakePercent = 50%`, then the ejector can eject 6 operators at once because `ejectableStakes = 50% * 10 stakes = 5 stakes + 1 stake` (for the additional ejected operator when the rate limit is reached).

However, if we update `ejectableStakes` after each ejection we get the following: after ejecting the first operator, the remaining stake is 9 stakes and `ejectableStakes = 50% * 9 stakes - 1 stake = 3.5 stakes`. After ejecting the second operator, the remaining stake is 8 stakes and `ejectableStakes = 50% * 8 stakes - 2 stakes = 2 stakes`. After the third ejection, the remaining stake is 7 stakes and the `ejectableStakes = 50% * 7 stakes - 3 stakes = 0.5 stakes`. Finally after the fourth ejection, the remaining stake is 6 stakes and `ejectableStakes` is hard set to 0.

So, clearly in this example the ejector was able to eject 6 operators instead of 4. Another way to view the issue is to notice the inconsistency between making a single call to `ejectOperators()` for a set of operators at once and comparing that to doing one call per operator.

L2	Ejector can eject one extra operator after the rate limit is reached	**RESOLVED**

Resolution:
A patch is applied to remove the code corresponding to ejecting an operator after the rate limit is reached in this PR.

The function `EjectionManager::ejectOperators()` allows an ejector to eject operators as long as the stake of the ejected operators does not exceed a rate limit during some specified period of time. However, when the check `stakeForEjection + operatorStake > amountEjectable` passes indicating that the rate limit is reached, the function still performs one more ejection before it breaks from the loop. This breaks the invariant that an ejector should not be able to eject operators with a total stake more than the configured limit.

```
if (
      isEjector[msg.sender]
      && quorumEjectionParams[quorumNumber].rateLimitWindow > 0
      && stakeForEjection + operatorStake > amountEjectable
) {
      ratelimitHit = true;

      stakeForEjection += operatorStake;
      ++ejectedOperators;

      // @Dedaub : Still eject even though the limit is hit.
      slashingRegistryCoordinator.ejectOperator(
      slashingRegistryCoordinator.getOperatorFromId(operatorIds[i][j]),
      abi.encodePacked(quorumNumber)
      );

      emit OperatorEjected(operatorIds[i][j], quorumNumber);

      break;
}
```

L3	`EjectionManager::ejectOperators()` does not emit `ratelimitHit = true` in an edge case	**ACKNOWLEDGED**

Comments:
The intended behavior is to emit true when the rate limit is exceeded and not when it is hit.

In the case where the following equality holds `takeForEjection + operatorStake = amountEjectable,` the function does not set `ratelimitHit` to true leading to a misleading emitted event.

L4	A registering operator with churn can bypass the `kickBIPsOfTotalStake` threshold condition	**ACKNOWLEDGED**

Comments:
The team acknowledges that this is not the desired behavior and will re-consider another approach.

When an operator registers with churn. The current implementation calculates the total amount of stakes (`totalStakes`) for the quorum by including both the registering operator's stake and the stake of the operator to be kicked out. For a churn to be valid, the kicked out operator must have stakes in that quorum less than `kickBIPsOfTotalStake * totalStakes`. However, since `totalStakes` includes the stakes of the registering operator, the latter can always put enough stakes to make this condition true for any existing operator and thus churn it out.

L5	Stale stakes are not prevented correctly in `BLSSignatureChecker::checkSignatures()`	**ACKNOWLEDGED**

Comments:
The code related for checking staled stakes is removed from the code in this PR. However, this does not solve the problem as stale stakes are still possible. In detail, the `checkSignature()` function does not provide guarantees that the returned totalStakes is correct as the amount of stakes read from the `StakeRegistry` might be outdated.

The team clarified this limitation in the docs and will reconsider a major change in the approach for future updates.

When the configuration `_staleStakesForbidden` is set, the function `BLSSignatureChecker::checkSignatures()` checks if the last update of the stakes of a given quorum happened with no longer than `MIN_WITHDRAWAL_DELAY_BLOCKS` blocks ago from the block that result is verified for. However, the function `checkSignatures()` allows checking the signature for any previous block which can be before the last update. An example of such case is given as the following:

Assume `MIN_WITHDRAWAL_DELAY_BLOCKS = 5` and `staleStakesForbidden=true`:

- **At block 0**: StakeRegistry is updated.
  - Operator A has 10 stakes
  - StakeRegistry storage: {A:10} {total :10}
- **At block 6**: stakers delegate 10 more stakes to operator A.
  - Operator A has now 20 stakes
  - StakeRegistry storage: {A:10} {total :10} **(not updated yet)**
- **At block 7**: operator B registered with 10 stakes.
  - Operator A has 20 stakes
  - Operator B has 10 stakes
  - StakeRegistry storage: {A:10, B:10} {total :20} **(not updated yet)**
- **At block 8**: StakeRegistry is updated.
  - Operator A has 20 stakes
  - Operator B has 10 stakes
  - StakeRegistry storage: {A:20, B:10} {total :30}

By calling `checkSignatures()` for **block 6** with a signature signed by operators A and B. The check will pass because the last update happened at **block 8** and clearly `8+5>6.` But in reality, the check should check for the last update that happened before

the requested block number (i.e., it should check for the last update before **block 6** which is at **block 0** which fails because `0+5<6`). With the current implementation this is not feasible because the storage variable `quorumUpdateBlockNumber` keeps track only of the time of the last update per quorum. Instead `quorumUpdateBlockNumber` should be modified to store the history of updates. Given that change, the condition can be updated to check that the latest quorum update **before the reference block number** happened within the last `MIN_WITHDRAWAL_DELAY_BLOCKS` from the reference block.

Another issue with this approach is that the requirement of the last stake update happened within no longer than `MIN_WITHDRAWAL_DELAY_BLOCKS` blocks does not provide any guarantees about stale stakes. This is because the delegation manager updates the operator stakes immediately when an unstaking operation is performed. Only the withdrawal of the stakes by the staker is delayed by `MIN_WITHDRAWAL_DELAY_BLOCKS` but the accounting for the operator stakes is calculated on the spot.

CENTRALIZATION ISSUES:

It is often desirable for DeFi protocols to assume no trust in a central authority, including the protocol's owner. Even if the owner is reputable, users are more likely to engage with a protocol that guarantees no catastrophic failure even in the case the owner gets hacked/compromised. We list issues of this kind below. (These issues should be considered in the context of usage/deployment, as they are not uncommon. Several high-profile, high-value protocols have significant centralization threats.)

ID	Description	STATUS
N1	Trust assumptions on permissioned roles	**INFO**

The EigenLayer-middleware contracts contain a number of operations performed by accounts with permissioned roles (owner, ejector, churn approver, etc). The protocol's security depends to an important extent on these accounts being trusted.

Note that we recommend performing critical checks on-chain to guarantee the desired system invariants instead of relying on off-chain checks. For instance, churning should be properly enforced on-chain (see M1, M2) instead of relying on the trusted churn approver.

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	Misleading errors	**RESOLVED**
The code has few misleading error messages: 1. `SlashingRegistryCoodinator._validateChurn::706` throws a `QuorumOperatorCountMismatch` error if the `kickParams.quorumNumber` and `quorumNumber` mismatch. The error name is misleading given there is no operator count mismatch. 2. `BLSApkRegistry::317` throws an `OnlyRegistryCoordinatorOwner` error when the check `msg.sender == address(registryCoordinator)` fails, however the check fails when the caller is not the `registryCoordinator`, not its owner.		
A2	Redundant storage operations in `StakeRegistry::updateOperatorsStake()`	**INFO**

The function `StakeRegistry::updateOperatorsStake()` updates the stakes of selected operators in selected quorums and checks if the stake of an operator falls below the minimum allowed stakes. If so, it sets their stake to zero in `operatorStakeHistory`, updates the total stakes of the quorum in `_totalStakeHistory`, and marks the operator for removal. However, the step of updating `operatorStakeHistory` and `_totalStakeHistory` is redundant and unnecessary. In particular, by marking the operator for removal, the only caller to this function, namely `SlashingRegistryCoordinator::_updateOperatorsStakes()` always deregisters the marked operators by calling `_kickOperator()`. Consequently, during the deregistration process, the `operatorStakeHistory` and `_totalStakeHistory` are updated and set to the correct values.		
A3	Insufficient check in `BLSApkRegistry::registerBLSPublicKey()`	**INFO**
The function `BLSApkRegistry::registerBLSPublicKey()` checks if the used `params.pubkeyG1` is not the neutral point by checking `require(pubkeyHash != ZERO_PK_HASH, ZeroPubKey())`. However, the function `hashG1Point()` hashes a point without reducing its coordinates. This means a neutral point represented with coordinates `(k*p, k*p)` for `k != 0` will result in a hash that is not the same as `ZERO_PK_HASH`. Therefore, the check will pass. Thats been said, this bug is not exploitable because later in the function, the operation `params.pubkeyG1.scalar_mul(gamma)` fails since `scalar_mul()` function uses the EVM pre-compiled function `ecMul` which reverts when given a point with non-reduced coordinates. It is recommended to either reduce the point before hashing it in `hashG1Point()` or revert if the coordinates are not given in the reduced format.		
A4	Point not on curve might be returned in `scalar_mul_tiny()`	**INFO**

The function `scalar_mul_tiny()` can return a point which is not on the curve if the scalar `s=1`. This happens because when `s=1`, the input point is returned as it is without any check if the point is on the curve. This bug cannot be exploited currently because the only location where this function is used is in `checkSignatures()` where the resulting point is added to the `apk` using the EVM precompiled function `ecAdd` which reverts when the point is not on the curve.

```
apk = apk.plus(
      params.nonSignerPubkeys[j].scalar_mul_tiny(
      BitmapUtils.countNumOnes(nonSigners.quorumBitmaps[j] &
signingQuorumBitmap)
      )
```

A5	Better implementation of `hashToG1()` algorithm	**INFO**

The function `hashToG1()` implements the try-and-increment algorithm for mapping a msg to `G1` point. This algorithm has non-deterministic runtime and can theoretically run for a large number of iterations. A constant time algorithm exists called Fouque-Tibouchi algorithm (Described here and here. Github implementation here) . The advantage of using a constant-time algorithm for hashing the point is to avoid significantly different gas costs based on the input. For example, the `checkSignatures()` maps the input message hash `msgHash` into a point. If the hash is controlled by an adversary, the function can potentially be subject to DoS attack by choosing malicious messages that require a lot of iterations to map them to a point. That's been said, with random messages, the average number of iterations for the try-and-increment algorithm is 2 which corresponds to an approximate gas cost of ~30k (~14k additional cost per iteration). On the other hand, the Fouque-Tibouchi algorithm has a constant gas cost size of ~140k. So, if the developers can guarantee that the hashed messages are always generated from a random oracle, it is a fair choice to keep using the try-and-increment algorithm.

A6	Inconsistent order of points when generating the coefficient `gamma` when checking the bilinear pairing	**INFO**

The bilinear pairing operation to verify the signatures involves the computation of a coefficient `gamma` derived from the public points and the signature. When `gamma` is computed for verifying the registration signature in `registerBLSPublicKey()` it uses a different point order from when it is computed to verify the signature of the committed result signature in `trySignatureAndApkVerification()`. In the first case, it is calculated as `gamma = h(sigma, P, P', H(m))` while in the second case it is computed as `gamma = h(msgHash, apk, apkG2, sigma)`. While this does not lead to any known security issue, it is recommended to keep the consistency.

A7	Redundant check in `SlashingRegistryCoordinator::registerOperator()`	**RESOLVED**

For the case of `NORMAL` registration type, the function `SlashingRegistryCoordinator::registerOperator()` performs a check that the number of operators for each quorum does not exceed the maximum allowed number of operators for that quorum. However, this check is already performed in the function `SlashingRegistryCoordinator::_registerOperator()` and thus it can be removed.

```
function registerOperator(...) external virtual override
onlyAllocationManager onlyWhenNotPaused(PAUSED_REGISTER_OPERATOR) {
      ...
      if (registrationType == RegistrationType.NORMAL) {
      uint32[] memory numOperatorsPerQuorum = _registerOperator({
            operator: operator,
            operatorId: operatorId,
            quorumNumbers: quorumNumbers,
            socket: socket,
            checkMaxOperatorCount: true
      }).numOperatorsPerQuorum;
```

```
        // For each quorum, validate that the new operator count does not
exceed the maximum
        // (If it does, an operator needs to be replaced -- see
`registerOperatorWithChurn`)
        // @Dedaub: unnecessary check because it is done in
_registerOperator()
        for (uint256 i = 0; i < quorumNumbers.length; i++) {
                uint8 quorumNumber = uint8(quorumNumbers[i]);

                require(
                numOperatorsPerQuorum[i] <=
_quorumParams[quorumNumber].maxOperatorCount,
                MaxOperatorCountReached()
                );
        }
        }
        ...
}
```

A8	Redundant check in `RegistryCoordinator::_kickOperator()`	**RESOLVED**

When kicking out an operator from a non-M2 quorum, the function `RegistryCoordinator::_kickOperator()` calls the overridden `_forceDeregisterOperator()` function which checks if the quorum is a non-M2 quorum. However, this check is unnecessary because `_forceDeregisterOperator()` is only called for non-M2 quorums. Therefore, an immediate call to the base function `super._forceDeregisterOperator()` is sufficient.

A9	Optimising operator ejection	**INFO**

Operator ejection, `EjectionManager::ejectOperators(..)`, can be optimised by keeping track of the quorums from which an operator should be ejected. Currently, there is a call for each operator and quorum-to-be-ejected-from pair to `SlashingRegistryCoordinator::ejectOperator(operator, quorumNumbers)`. Instead, by collecting the set of quorums an operator should be ejected from, one can eject the operator from these quorums in one go with one call to the above function. This will help avoid some gas costs (in particular by reducing the number of function calls, and storage access/update).

A10	Optimization for `PermissionController::setAppointee()`	**INFO**

The function `PermissionController::setAppointee()` checks if an appointee is already included in the set by checking the following requirement: `require(!permissions.appointeePermissions[appointee].contains(targetSelector), AppointeeAlreadySet()).` It follows that by adding `targetSelector` to the set using `permissions.appointeePermissions[appointee].add(targetSelector)`. An optimization is possible by simple removing the `require` statement and instead modify the second statement to `require(permissions.appointeePermissions[appointee].add(targetSelector), AppointeeAlreadySet()).`

A similar optimization can be applied for the function `PermissionController::removeAppointee()`.

A11	Incorrect comments and text in the code and in the docs	**RESOLVED**

The code has few inaccurate text comments:

- The comment in `BLSApkRegistry::registerBLSPublicKey()`:
  `// e(sigma + P * gamma, [-1]_2) = e(H(m) + [1]_1 * gamma, P')`

should be:
`// e(sigma + P * gamma, [1]_2) = e(H(m) + [1]_1 * gamma, P')`

- The comment in `StakeRegistry::updateOperatorsStake()` about `quorumsToRemove` should be updated as `quorumsToRemove` seems to be a deprecated variable.
- The comment above the function `BN254::pairing()` is inaccurate because the function supports only the check of the product of 2 pairing and not n pairings. Also the given example is wrong as it should be "*`pairing(P1(), P2(), P1.negate(), P2())` should return true.*"

In addition, the readme docs has some inaccurate texts:

- In the file `StakeRegistry.md` section `#### updateOperatorStake` mentions that this method is ONLY callable by the `RegistryCoordinator` however it should say that it has not access control restrictions.
- In the file `AllocationManager.md` section `## Parameterization`, the description of `ALLOCATION_CONFIGURATION_DELAY` is wrong and it should be *"The delay in blocks before allocation delay change takes effect"*.

A12	Compiler bugs	**INFO**

The code is compiled with Solidity `0.8.27`. Version `0.8.27`, in particular, has no known bugs.

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.

APPENDIX

Description
Proof of Concept - M2
This test uses the deployment mocked contracts from the test suite in the project. The following test function should be placed in the file `RegistryCoordinatorUnit.t.sol`.

```
function test_registerOperatorWithChurn_BypassMaxOperatorCount(
    uint256 pseudoRandomNumber
) public {
    bytes memory quorumNumbers = new bytes(1);
    quorumNumbers[0] = bytes1(0);
    ISignatureUtilsMixinTypes.SignatureWithSaltAndExpiry memory emptyAVSRegSig;

    uint32 kickRegistrationBlockNumber = 100;

    uint256 quorumBitmap = BitmapUtils.orderedBytesArrayToBitmap(quorumNumbers);

    cheats.roll(kickRegistrationBlockNumber);

    // register all operators until the quorum is full
    for (uint256 i = 0; i < defaultMaxOperatorCount; i++) {
        BN254.G1Point memory pubKey =
            BN254.hashToG1(keccak256(abi.encodePacked(pseudoRandomNumber, i)));
        address operator = _incrementAddress(defaultOperator, i);

        _registerOperatorWithCoordinator(operator, quorumBitmap, pubKey);
    }

    // prepare the operator to register
    address operatorToRegister = _incrementAddress(defaultOperator,
defaultMaxOperatorCount);
    BN254.G1Point memory operatorToRegisterPubKey =
```

```
        BN254.hashToG1(keccak256(abi.encodePacked(pseudoRandomNumber,
defaultMaxOperatorCount)));
    bytes32 operatorToRegisterId = BN254.hashG1Point(operatorToRegisterPubKey);
    blsApkRegistry.setBLSPublicKey(operatorToRegister, operatorToRegisterPubKey);
    _setOperatorWeight(operatorToRegister, defaultQuorumNumber, defaultStake);


    // Use empty operatorKickParams.
    // This means the address of the operator to kick is 0 which refers to a
non-existing operator.
    ISlashingRegistryCoordinatorTypes.OperatorKickParam[] memory operatorKickParams
=
        new ISlashingRegistryCoordinatorTypes.OperatorKickParam[](1);


    ISignatureUtilsMixinTypes.SignatureWithSaltAndExpiry memory signatureWithExpiry
=
    _signOperatorChurnApproval(
        operatorToRegister,
        operatorToRegisterId,
        operatorKickParams,
        defaultSalt,
        block.timestamp + 10
    );
    cheats.prank(operatorToRegister);

    // register with churn using an non-existing operator to kick
    registryCoordinator.registerOperatorWithChurn(
        quorumNumbers,
        defaultSocket,
        pubkeyRegistrationParams,
        operatorKickParams,
        signatureWithExpiry,
        emptyAVSRegSig
    );

    // The total number of operators exceeded the max number
    uint32 count = indexRegistry.totalOperatorsForQuorum(0);
    assertEq(count , defaultMaxOperatorCount + 1);
}
```

Proof of Concept - M1

This test uses the deployment mocked contracts from the test suite in the project. The following test function should be placed in the file `SlashingRegistryCoordinatorUnit.t.sol` under the contract `SlashingRegistryCoordinator_RegisterWithChurn`.

```
function test_registerOperatorWithChurn_BypassMaxOperatorCount2() public {

    // register an operator "operatorToKick2" to quorum 0
    Operator memory operatorToKick2 = operatorsByID[operatorIds.at(5)];
    bytes32 operatorToKickId2 = operatorIds.at(5);

    uint32[] memory operatorSetIds = new uint32[](1);
    operatorSetIds[0] = 0;
    registerOperatorInSlashingRegistryCoordinator(operatorToKick2, "socket:8545",
operatorSetIds);

    _setOperatorWeight(testOperator.key.addr, registeringStake);
    _setOperatorWeight(operatorToKick2.key.addr, operatorToKickStake);

    ISlashingRegistryCoordinator.OperatorInfo memory operatorInfoBefore =
        slashingRegistryCoordinator.getOperator(testOperator.key.addr);
    assertEq(
        uint256(operatorInfoBefore.status),
        uint256(ISlashingRegistryCoordinatorTypes.OperatorStatus.NEVER_REGISTERED),
        "Registering operator should have NEVER_REGISTERED status"
    );

    ISlashingRegistryCoordinator.OperatorInfo memory kickedOperatorInfoBefore =
        slashingRegistryCoordinator.getOperator(operatorToKick2.key.addr);
    assertEq(
        uint256(kickedOperatorInfoBefore.status),
        uint256(ISlashingRegistryCoordinatorTypes.OperatorStatus.REGISTERED),
        "Kicked operator should be REGISTERED"
    );
```

```
    uint192 kickedBitmapBefore =
slashingRegistryCoordinator.getCurrentQuorumBitmap(operatorToKickId2);
    assertEq(kickedBitmapBefore, uint192(1), "Kicked operator should be registered
to quorum 0 only");

    // prepare the operatorKickParams in a way that we to kick "operatorToKick2"
from quorum 1 (which he is not actually registered to)
    ISlashingRegistryCoordinatorTypes.OperatorKickParam[] memory operatorKickParams
= _createOperatorKickParams(operatorToKick2.key.addr, quorumNumbers);

    ISignatureUtilsMixinTypes.SignatureWithSaltAndExpiry memory
churnApproverSignature = _signChurnApproval(
        testOperator.key.addr, testOperatorId, operatorKickParams, defaultSalt,
defaultExpiry
    );

    _registerOperatorWithChurn(
        testOperator, operatorKickParams, "socket:8545", churnApproverSignature
    );

    ISlashingRegistryCoordinator.OperatorInfo memory operatorInfoAfter =
        slashingRegistryCoordinator.getOperator(testOperator.key.addr);
    assertEq(
        uint256(operatorInfoAfter.status),
        uint256(ISlashingRegistryCoordinatorTypes.OperatorStatus.REGISTERED),
        "Registering operator should have REGISTERED status"
    );

    ISlashingRegistryCoordinator.OperatorInfo memory kickedOperatorInfoAfter =
        slashingRegistryCoordinator.getOperator(operatorToKick2.key.addr);
    assertEq(
        uint256(kickedOperatorInfoAfter.status),
        uint256(ISlashingRegistryCoordinatorTypes.OperatorStatus.REGISTERED),
        "Registering operator should still have REGISTERED status"
    );

    uint192 currentBitmap =
slashingRegistryCoordinator.getCurrentQuorumBitmap(testOperatorId);
    assertEq(currentBitmap, uint192(2), "Registered operator should be registered
in quorum 1 now");
```

```
    uint192 kickedBitmap =
slashingRegistryCoordinator.getCurrentQuorumBitmap(operatorToKickId2);
    assertEq(kickedBitmap, uint192(1), "Kicked operator should still be in quorum
0");

    // The total number of operators exceeded the max number
    uint32 count = indexRegistry.totalOperatorsForQuorum(uint8(quorumNumbers[0]));
    assertEq(count ,
slashingRegistryCoordinator.getOperatorSetParams(uint8(quorumNumbers[0])).maxOpera
torCount + 1, "The max number of operators for quorum should be exceeded");
}
```