yAudit

April 2026

Prepared for
Den

Audited by
adriro
fedebianu

Den MLS Wallet

Smart Contract Security Assessment

Contents

1 Review Summary 3
- 1.1 Protocol Overview . . . 3
- 1.2 Audit Scope . . . 3
- 1.3 Risk Assessment Framework . . . 5
  - 1.3.1 Severity Classification . . . 5
- 1.4 Key Findings . . . 6
- 1.5 Overall Assessment . . . 6

2 Audit Overview 6
- 2.1 Project Information . . . 6
- 2.2 Audit Timeline . . . 6
- 2.3 Audit Resources . . . 6
- 2.4 Critical Findings . . . 8
- 2.5 High Findings . . . 8
- 2.6 Medium Findings . . . 8
  - 2.6.1 Transaction recovery address cannot be revoked, allowing a compromised one to drain all accounts after timelock . . . 8
  - 2.6.2 Contract interaction policies cannot constrain the attached ETH value . . 10
  - 2.6.3 Address OneOf Merkle proof is embedded in the constraints hash, limiting the constraint to a single value . . . 11
- 2.7 Low Findings . . . 12
  - 2.7.1 `TransactionType.Any` policies cannot cap token-transfer amounts via rate limit . . . 12
  - 2.7.2 Batch atomicity relies on Guardian behavior rather than signed batch intent 14
- 2.8 Gas Savings Findings . . . 15
  - 2.8.1 Unnecessary Array and Struct type check in `_isParameterAllowedByConstraint()` . . . 15
  - 2.8.2 Redundant `isGroup()` check before `isGroupMember()` . . . 16
  - 2.8.3 Redundant validation checks in `OrganizationAccountFactoryBase.implementation()` . . . 17
- 2.9 Informational Findings . . . 17
  - 2.9.1 Redundant chainId and organization fields in EIP-712 struct hashes . . . 17
  - 2.9.2 Guardian signature validation accepts any enabled Safe module . . . 18
  - 2.9.3 BatchedTransaction does not validate well-formedness of packed payload . 19
  - 2.9.4 `SafeExecutorModule` does not restrict call targets to known organizations 20
  - 2.9.5 `OrganizationPolicyBase.setPolicies()` relies on off-chain layers to ensure `newPoliciesRoot` matches the policy dump . . . 21
  - 2.9.6 Pending signatures are not bound to a policy root or policy version . . . 22
  - 2.9.7 Rate limits and amount thresholds are unreliable for anyToken policies . . 23
  - 2.9.8 Dual initialization checks may cause confusion . . . 24
  - 2.9.9 `getActualDestination()` implementation could be simplified . . . 25
  - 2.9.10 Cross-chain organization address stability depends on initial implementation address . . . 26
  - 2.9.11 Dirty upper bytes not cleared in `_extractContractInnerSignature()` . 27
  - 2.9.12 Missing external getters for internal state queries . . . 27
  - 2.9.13 Unused `AccountTransactionRejection` enum value . . . 28
  - 2.9.14 Add `onlyProxy` modifier to `OrganizationImplementation.upgradeToAndCallWithAuthorization()` . . . 29

2.9.15 `SafeExecutorModule` documents an EOA executor but does not enforce it onchain 30
2.9.16 Strengthen whitelist proxy initialization 31
2.10 Final Remarks 31

1 Review Summary

1.1 Protocol Overview

Den MLS Wallet is a policy-based non-custodial wallet infrastructure for organizations. Assets are held in Account contracts owned by Organization contracts, while authorization is enforced through members, groups, admins, Merkle-rooted policies, Guardian validation, ERC-1271 account signatures, Safe module execution, and timelocked recovery flows.

1.2 Audit Scope

This audit covers 67 smart contracts and libraries totaling approximately 3,540 lines of code across 15 days of review.

```
src/
├── account
│   ├── AccountImplementation.sol
│   ├── AccountProxy.sol
│   └── libraries
│       └── storage
│           └── LibAccountOrganizationAddressStorage.sol
├── implementation-whitelist
│   ├── ImplementationWhitelistImplementation.sol
│   ├── ImplementationWhitelistProxy.sol
│   └── libraries
│       └── storage
│           └── LibImplementationWhitelistStorage.sol
├── interfaces
│   ├── IAccount.sol
│   ├── IBatchedTransaction.sol
│   ├── IImplementationWhitelist.sol
│   ├── IOrganization.sol
│   ├── IOrganizationFactory.sol
│   ├── ISafeExecutorModule.sol
│   └── organization
│       ├── IOrganizationAccountFactory.sol
│       ├── IOrganizationAccountSignature.sol
│       ├── IOrganizationAccountTransaction.sol
│       ├── IOrganizationAdmin.sol
│       ├── IOrganizationAdminOperationTimelock.sol
│       ├── IOrganizationGroups.sol
│       ├── IOrganizationGuardian.sol
│       ├── IOrganizationGuardianRecovery.sol
│       ├── IOrganizationInitialization.sol
│       ├── IOrganizationMembers.sol
│       ├── IOrganizationPolicy.sol
│       ├── IOrganizationSignatures.sol
│       └── IOrganizationTxRecovery.sol
├── libraries
```

```
│   ├── BytesUtils.sol
│   ├── ContractInteractionUtils.sol
│   ├── MerkleUtils.sol
│   ├── SignatureUtils.sol
│   ├── TimelockUtils.sol
│   └── TokenTransferUtils.sol
├── organization
│   ├── OrganizationFactory.sol
│   ├── OrganizationImplementation.sol
│   ├── OrganizationProxy.sol
│   ├── base
│   │   ├── OrganizationAccountFactoryBase.sol
│   │   ├── OrganizationAccountSignatureBase.sol
│   │   ├── OrganizationAccountTransactionBase.sol
│   │   ├── OrganizationAdminBase.sol
│   │   ├── OrganizationAdminOperationTimelockBase.sol
│   │   ├── OrganizationGroupsBase.sol
│   │   ├── OrganizationGuardianBase.sol
│   │   ├── OrganizationGuardianRecoveryBase.sol
│   │   ├── OrganizationInitializationBase.sol
│   │   ├── OrganizationMembersBase.sol
│   │   ├── OrganizationPolicyBase.sol
│   │   ├── OrganizationSignaturesBase.sol
│   │   └── OrganizationTxRecoveryBase.sol
│   ├── common
│   │   └── OrganizationModifiers.sol
│   └── libraries
│       ├── LibOrganizationAccountFactory.sol
│       ├── LibOrganizationAccountSignature.sol
│       ├── LibOrganizationAccountTransaction.sol
│       ├── LibOrganizationAdmin.sol
│       ├── LibOrganizationAdminOperationTimelock.sol
│       ├── LibOrganizationEIP712.sol
│       ├── LibOrganizationGroups.sol
│       ├── LibOrganizationGuardian.sol
│       ├── LibOrganizationGuardianRecovery.sol
│       ├── LibOrganizationInitialization.sol
│       ├── LibOrganizationMembers.sol
│       ├── LibOrganizationPolicy.sol
│       ├── LibOrganizationSignatures.sol
│       ├── LibOrganizationTxRecovery.sol
│       ├── policy
│       │   ├── LibPolicyApproval.sol
│       │   ├── LibPolicyContractInteraction.sol
│       │   ├── LibPolicyDestination.sol
│       │   ├── LibPolicyInitiator.sol
│       │   ├── LibPolicyParameterConstraints.sol
│       │   ├── LibPolicyRateLimits.sol
│       │   └── LibPolicyTokenTransfer.sol
│       └── storage
│           ├── LibOrganizationAccountFactoryStorage.sol
```

```
│           ├── LibOrganizationAdminOperationTimelockStorage.sol
│           ├── LibOrganizationAdminStorage.sol
│           ├── LibOrganizationDeployerAddressStorage.sol
│           ├── LibOrganizationGroupsStorage.sol
│           ├── LibOrganizationGuardianStorage.sol
│           ├── LibOrganizationMembersStorage.sol
│           ├── LibOrganizationPolicyStorage.sol
│           ├── LibOrganizationRecoveryStorage.sol
│           ├── LibOrganizationSignaturesStorage.sol
│           └── LibOrganizationUpgradeStorage.sol
├── safe-module
│   ├── BatchedTransaction.sol
│   └── SafeExecutorModule.sol
└── types
    ├── AdminTypes.sol
    ├── CommonTypes.sol
    ├── PolicyTypes.sol
    └── RecoveryTypes.sol
```

1.3 Risk Assessment Framework

1.3.1 Severity Classification

Severity	Description	Potential Impact
Critical	Immediate threat to user funds or protocol integrity	Direct loss of funds, protocol compromise
High	Significant security risk requiring urgent attention	Potential fund loss, major functionality disruption
Medium	Important issue that should be addressed	Limited fund risk, functionality concerns
Low	Minor issue with minimal impact	Best practice violations, minor inefficiencies
Undetermined	Findings whose impact could not be fully assessed within the time constraints of the engagement. These issues may range from low to critical severity, and although their exact consequences remain uncertain, they present a sufficient potential risk to warrant attention and remediation.	Varies based on actual severity
Gas	Findings that can improve the gas efficiency of the contracts.	Increased transaction costs
Informational	Code quality and best practice recommendations	Reduced maintainability and readability

Table 1: severity classification

1.4 Key Findings

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	0
Medium	3
Low	2
Informational	16

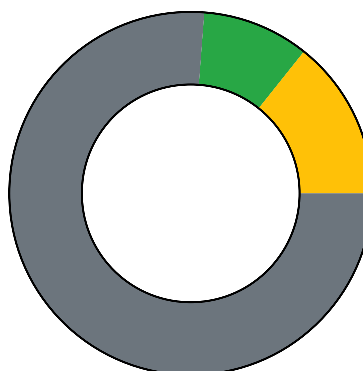


Figure 1: Distribution of security findings by impact level

1.5 Overall Assessment

The codebase is modular and well documented, with clear separation between Organization logic, Account execution, implementation whitelisting, Safe module support, shared libraries, typed data signatures, and namespaced storage. No critical or high severity issues were identified. The findings concentrate on edge cases in policy validation, rate-limit accounting, recovery and timelock flows, and the trust assumptions placed on the Guardian and Safe execution paths.

2 Audit Overview

2.1 Project Information

Protocol Name: Den
Repository: https://github.com/OnChainDen/mls-contracts
Commit URLs:

- Review commit: 91d2b295bb20f1a972099f2f4b8325c1af67678c
- Final commit: 84ea74860ba19dbc01db888bd510a558405bb914

2.2 Audit Timeline

The audit was conducted from April 9 to 29, 2026.

2.3 Audit Resources

- Code repository
- Detailed documentation

Category	Mark	Description
Access Control	Good	Roles for admins, members, groups, the Guardian, and the recovery addresses are clearly separated and consistently enforced through dedicated modifiers.
Mathematics	Good	The system relies on simple accounting for nonces, rate-limit windows, and timelocks, with no mathematical issues identified during the review.
Complexity	Good	The protocol spans many interacting modules, including organization base contracts, policy validation, Merkle-rooted allowlists, Safe module execution, and recovery timelocks. Several findings stem from edge cases where these layers overlap, such as policy value constraints, rate-limit accounting under TransactionType.Any, and the embedding of Merkle proofs inside the constraints hash.
Libraries	Good	The codebase makes appropriate use of OpenZeppelin contracts for proxies, initialization, ownership, and Merkle proof verification, and integrates with the Safe contracts for module execution.
Decentralization	Average	By design, the Guardian, organization admins, and recovery addresses hold significant powers over account execution, upgrades, and recovery flows.
Documentation	Good	The contracts are well documented with NatSpec, and detailed high-level documentation covering the policy system, Guardian protection, and deployment flow was provided to the auditors.
Testing and verification	Good	The codebase ships with a comprehensive test suite covering the organization, account, policy, recovery, and Safe module components.

Table 2: Code Evaluation Matrix

2.4 Critical Findings

None.

2.5 High Findings

None.

2.6 Medium Findings

2.6.1 Transaction recovery address cannot be revoked, allowing a compromised one to drain all accounts after timelock

Technical Details

The transaction-and-ERC1271 recovery address is set exactly once and never modified or cleared throughout the contract's lifetime. The only assignment to `txRecovery.recoveryAddress` is in `LibOrganizationTxRecovery.initializeTxRecovery()`, guarded by `_validateTxRecoveryNotConfiguredOrRevert()` which reverts when `recoveryAddress != address(0)`. After the first successful initialization (either at deployment via `LibOrganizationInitialization.initialize` or post-deployment via `finalizeInitializeTxRecovery`), no code path can update or clear that field. `disableTransactionAndERC1271Recovery()` only flips `isEnabled = false` and clears `pendingEnableTimestamp`; it does not touch `recoveryAddress` or `timelockDurationSeconds`. As a result, `_validateTxRecoveryNotConfiguredOrRevert()` keeps reverting forever, so neither admins nor the recovery address itself can re-initialize with a different address.
The capabilities exclusively held by `recoveryAddress` are gated by `onlyTxRecoveryAddress` in `OrganizationTxRecoveryBase`:

- `initiateEnableTransactionAndERC1271Recovery()` — starts the enable timelock
- `finalizeEnableTransactionAndERC1271Recovery()` — flips `isEnabled = true` once the timelock elapses
- `cancelEnableTransactionAndERC1271Recovery()` — cancels a pending enable
- `disableTransactionAndERC1271Recovery()` — instantly disables (does not clear the address)
- `executeRecoveryAccountTransaction(account, to, value, data)` — performs an arbitrary `call` on any organization account, bypassing guardian, policy, signature, nonce, and rate-limit checks (`nonce=0, policyId=0` at `OrganizationTxRecoveryBase.sol#L59`)

There is no admin-authorized path for any of the following actions:

- Replace the recovery address with a fresh key
- Clear the recovery address (forcing a fresh initialization)
- Cancel an enable that the recovery address itself has initiated (the cancel is also gated to the recovery address)

The same one-time initialization pattern also applies to `guardianRecovery.recoveryAddress`: it is assigned in `LibOrganizationGuardianRecovery.initializeGuardianRecovery()` and guarded by `_validateGuardianRecoveryNotConfiguredOrRevert()`, so it cannot be rotated or

cleared after initial configuration. The impact is different and smaller than tx/ERC1271 recovery: a compromised `guardianRecoveryAddress` cannot execute arbitrary account transactions or sign tx-recovery ERC-1271 payloads, but it can unilaterally drive the guardian recovery flow and install an attacker-controlled guardian after the timelock.

Impact

Medium. If the `transactionAndERC1271RecoveryAddress` is ever compromised, the attacker has a one-shot path to drain every account ever deployed by the organization, and the legitimate operators have no on-chain remediation:

1. If recovery is currently disabled, the attacker calls `initiateEnableTransactionAndERC1271Recovery()`. The timelock begins.
2. Admins notice and want to stop it. They cannot: `cancelEnable...` and `disable...` are both `onlyTxRecoveryAddress`, and there is no admin path to overwrite or clear the address.
3. After the configured timelock elapses, the attacker calls `finalizeEnableTransactionAndERC1271Recovery()`, then `executeRecoveryAccountTransaction()` for every account in turn, draining all assets.

The only mitigation available to admins is to race the attacker by moving funds out of every account through the normal `executeAccountTransaction` flow within the timelock window. That requires:

- The compromise to be detected promptly
- Guardian + admin signature collection within the timelock window
- A destination policy that authorizes external withdrawal (which may not exist for assets the org normally holds long-term)
- Sufficient time and gas to drain every deployed account

In effect, the timelock is the only friction between a compromised recovery key and full asset loss, with no admin-side override. This converts the recovery address into a single point of failure that is structurally more powerful than the guardian/admin set combined.
The same address also signs the recovery path of `Account.isValidSignature()` via the `0x00` signature-type prefix (see `LibOrganizationAccountSignature._validateRecoverySignature`), so a compromised key can additionally forge ERC-1271 signature approvals against any integrator that trusts the account.

Recommendation

Add a guardian + admin-threshold operation that fully clears the tx recovery state so a fresh `initiateInitializeTransactionAndERC1271Recovery()` can be used to configure a new recovery address through the existing deferred initialization flow. The clear should reset `txRecovery.recoveryAddress`, `timelockDurationSeconds`, `isEnabled`, `pendingEnableTimestamp`, and any pending initialization fields.

Developer Response

Implemented the suggested change and added a single clear function that clears the recovery state for both guardian and transaction recovery.

The clear function is admin-authorized and guardian-gated, and deliberately **not** timelocked, so that admins can immediately clear a compromised recovery address before the recovery timelock wait-period has ended and an attacker is able to use the recovery address.
PR: #153 Commit in main: 84ea748

2.6.2 Contract interaction policies cannot constrain the attached ETH value

Policies configured for `ContractInteractions` cannot restrict the native ETH value attached to a call, allowing transactions to payable functions with arbitrary `msg.value`.

Technical Details

In `LibOrganizationPolicy.isTransactionAllowedByPolicy()`, the `ContractInteractions` branch filters out token transfers but does not validate the `value` field against any policy constraint before delegating to `LibPolicyContractInteraction.isContractInteractionAllowedByPolicy()`:

```
1 if (proofs.policy.config.transactionType == TransactionType.ContractInteractions) {
2     if (TokenTransferUtils.isTransactionTokenTransfer(data, value)) return false;

4     return LibPolicyContractInteraction.isContractInteractionAllowedByPolicy({...});
5 }
```

A native ETH transfer (empty calldata with nonzero value) is already classified as a token transfer and filtered out. However, a call with nonempty calldata and a positive `value` passes the token transfer check and enters the contract interaction path. The policy system validates the function selector, destination, and parameter constraints, but has no mechanism to express a constraint on the attached `value`. This means a policy that authorizes a call to a payable function implicitly allows any amount of ETH to be sent along with it.

Impact

Medium. Organizations cannot limit the ETH value attached to authorized contract interactions. A member authorized to call a specific payable function could drain the account's ETH balance in a single call, even if the policy was only intended to authorize the function's logic without large value transfers.

Recommendation

Add an optional value constraint to the contract interaction policy configuration, allowing administrators to set a maximum or exact ETH value that can be attached to authorized calls.

Developer Response

This finding has been addressed in PR 128, which was merged into main in commit f8e0673. Policies for contract interactions must now explicitly specify the maximum amount of native ETH that can be sent as part of a contract call.

2.6.3 Address OneOf Merkle proof is embedded in the constraints hash, limiting the constraint to a single value

The `OneOf` constraint type for address parameters is intended to allow any address from a Merkle-tree-based allowlist. However, the Merkle proof required for verification is included inside the constraints data that gets hashed into the allowed functions Merkle tree, effectively locking the constraint to a single provable address.

Technical Details

In `LibPolicyContractInteraction._isFunctionAllowedByPolicy()`, the constraints bytes are hashed to produce the `constraintsHash`, which is then combined with the function selector to form the leaf that must exist in the allowed functions Merkle tree:

```
bytes32 constraintsHash = keccak256(constraints);
bytes32 funcLeaf = _computeFunctionLeaf(selector, constraintsHash);
return MerkleProof.verify(functionProof, policy.roots.allowedFunctionsRoot, funcLeaf);
```

The `constraints` bytes are an ABI-encoded array of `ParameterConstraint` structs. Each `ParameterConstraint` contains a `paramValueInListProof` field, which holds the Merkle proof used by `_isAddressParameterAllowedByConstraint()` to verify that the actual address value is a member of the allowed addresses tree.
Because `paramValueInListProof` is part of the ABI-encoded constraints, and the hash of the full constraints is committed in the allowed functions Merkle tree, only one specific proof (and therefore one specific address from the allowlist) can satisfy both the function-level Merkle verification and the address-level Merkle verification at the same time. Submitting a different proof for a different allowed address would change the constraints hash, causing the function-level Merkle verification to fail.
This makes the `OneOf` constraint behave identically to an `Exact` constraint, defeating its purpose of allowing any address from a predefined set.

Impact

Medium. Policies that use `OneOf` address constraints for contract interaction parameters will silently restrict transactions to a single address instead of the full allowlist. This limits the expressiveness of the policy system and may force administrators to create redundant policies for each allowed address.

Recommendation

Move the `paramValueInListProof` outside of the data that gets hashed into the allowed functions Merkle tree. One approach is to pass the proofs as a separate parameter alongside the constraints, so that changing the proof does not affect the constraints hash used for function-level verification.

Developer Response

This finding has been addressed in PR #130, which was merged into main in commit ae927f4. Proofs for parameters with `OneOf` constraints were moved out of `struct ParameterConstraint` and into `struct ValidationProofs`.

Note: the proofs are now represented as type `bytes` , not `bytes32[][]` , to prevent `OrganizationImplementation` from going over the 24KB EIP-170 limit. The `bytes` are decoded into the nested double array in the dynamically linked policy libraries, which have bytecode headroom to spare.

2.7 Low Findings

2.7.1 `TransactionType.Any` policies cannot cap token-transfer amounts via rate limit

Technical Details

Rate-limit accounting for an executed transaction is handled in `LibOrganizationAccountTransaction._validateAndUpdateRateLimitOrRevert()`. The branch that decides how much usage to attribute to the current transaction is:

```
if (policy.config.transactionType == TransactionType.TokenTransfers) {
    usageAmount = TokenTransferUtils.extractTransferAmount(data, params.value);
} else {
    usageAmount = 1; // Count-based limit for non-transfer transactions
}
```

The selector is purely the policy's `transactionType` enum — the shape of the calldata is not consulted. Any policy whose `transactionType` is not exactly `TokenTransfers` gets count-based accounting (`usageAmount = 1`) even when the calldata is a real ERC-20 `transfer(address,uint256)` / `transferFrom(address,address,uint256)` that the codebase can already recognize via `TokenTransferUtils.isTransactionTokenTransfer()` (used for routing in `LibOrganizationPolicy.isTransactionAllowedByPolicy()` and `LibOrganizationPolicy.sol:138`).
The concrete case this affects is `TransactionType.Any`. An `Any` policy authorizes the transaction at `LibOrganizationPolicy.sol:153-160` purely on destination matching. The rate-limit surface, however, is a separate layer that runs *after* the policy check and does have access to `data` and `value`. Choosing to ignore calldata shape at this layer is an active decision, not a consequence of "no common field" — it produces a second-order asymmetry:

- The same `policy.config.rateLimit.timeIntervalLimit` `uint256` scalar means "N calls per window" when the operator uses `Any` and "N units of amount per window" when they use `TokenTransfers`. The field is semantically overloaded purely by the sibling `transactionType` field.
- An operator who uses `Any` because they want to cover both shapes of call behind one policy loses the ability to express amount-based caps on the transfer half of the traffic. The only workaround is to split into one `TokenTransfers` policy plus one `ContractInteractions` policy, which forces a different set of tradeoffs (two policy leaves, two separate approval/timelock flows for admins).
- An operator setting `timeIntervalLimit = 1000` on an `Any` policy while mentally translating "1000 USDC per day" is silently given "1000 calls per day, each of unbounded amount". This is a configuration footgun that is not flagged by the type system or by documentation.

Impact

Low. Operators who use `TransactionType.Any` with rate limiting cannot express any amount-based cap for the transfer-shaped traffic that passes through the policy.
This is most likely to affect:

- Operations teams who migrate a policy from `TokenTransfers` to `Any` to also allow occasional contract calls, keeping the same `timeIntervalLimit` value. The intent "amount cap" silently becomes "count cap".
- Operations teams who design a single `Any` policy for a DeFi workflow that mixes `approve` / `transfer` / router calls under one rate cap, expecting a single USD-value budget per window.

Recommendation

Split the single `timeIntervalLimit` scalar in `RateLimitConfig` into two independent caps, one per accounting dimension, and accrue each from calldata shape:

```
struct RateLimitConfig {
    RateLimitType limitType;
    uint16 timeIntervalHours;
    uint256 timeIntervalCountLimit;   // 0 = disabled; bucket always accrues +1 per call
    uint256 timeIntervalAmountLimit;  // 0 = disabled; bucket accrues transfer amount when calldata is transfer-shaped
    uint256 anchorTimestamp;
    RateLimitScope initiatorScope;
    RateLimitScope sourceScope;
    RateLimitScope destinationScope;
}
```

Accounting in `_validateAndUpdateRateLimitOrRevert` becomes shape-driven rather than type-driven:

1. If `timeIntervalCountLimit > 0`, accrue `+1` to the count bucket and revert if the bucket would exceed the limit.
2. If `timeIntervalAmountLimit > 0` and `TokenTransferUtils.isTransactionTokenTransfer(data, value)` is true, accrue `TokenTransferUtils.extractTransferAmount(data, value)` to the amount bucket and revert if the bucket would exceed the limit.
3. Both checks are independent; a transaction must pass both to execute. The two buckets should have separate storage keys (e.g., a `(usageKey, bucketKind, timeWindow)` triple) to avoid aliasing.

After this change:

- `Any` policies can express "max N calls per window AND max M transferred per window" in one policy leaf.
- `TokenTransfers` policies can additionally cap call count (defense-in-depth against many small transfers).
- `ContractInteractions` policies can leave `timeIntervalAmountLimit = 0` and keep today's count-based semantics unchanged.
- The semantic overload on the old `timeIntervalLimit` ("meaning depends on `transactionType`") is eliminated: each field has one, explicit unit.

This is a breaking change to `RateLimitConfig`, `PolicyConfig`, and the policy Merkle leaf. If the team prefers to keep the existing layout, the minimum alternative is to document explicitly in `docs/MERKLETREE_ARCHITECTURE.md` (or the user-facing policy-builder docs) that under

`TransactionType.Any`, `timeIntervalLimit` is always a call-count cap and is never applied to transfer amounts, so that operators do not rely on the amount-cap semantic they are used to from `TransactionType.TokenTransfers`.

Developer Response

Acknowledged.
It was an intentional design decision to make it so that if a policy has `TransactionType.Any` + a rate limit specified, then:

- the rate-limit is a count-based rate limit, regardless of whether or not the transaction is a token transfer or a contract interaction
- the rate-limit is *not* an amount-based rate limit if the transaction is a token transfer

This is because if a policy has `TransactionType.Any` , the user will only be able configure fields for the policy that are shared by both token transfers and contract interactions. Since contract interaction policies don't have amount-based rate-limits, the lowest common denominator for both transaction types is count-based rate limits.
We'll keep the existing layout and will update documentation and comments in the code to explicitly document that under `TransactionType.Any`, `timeIntervalLimit` is always a call-count cap and is never applied to transfer amounts.
Updated documentations and comments in code to explicitly document that for `TransactionType.Any` policies, `timeIntervalLimit` is always a call-count cap and is never applied to transfer amounts: - PR: #138 - Commit in `main`: 8baca17

2.7.2 Batch atomicity relies on Guardian behavior rather than signed batch intent

The Guardian security model relies on `BatchedTransaction` to execute groups of operations atomically, but the user authorization layer signs each operation independently. This means batch cohesion is operationally enforced by the Guardian path rather than cryptographically enforced across the full stack.

Technical Details

`SafeExecutorModule.executeOnBehalf()` enables the authorized executor to route calls through `BatchedTransaction.execute()`, which gives the Guardian Safe an atomic all-or-nothing submission path. The documentation explicitly treats this batching behavior and the associated full-revert semantics as part of the Guardian security model.
At the same time, the organization and wallet layers expose single-operation execution primitives such as `OrganizationAccountTransactionBase.executeAccountTransaction()` and `AccountImplementation.executeTransaction()`. The signed payloads for account transactions and admin operations authorize each operation on its own. They do not commit to a shared batch hash, batch identifier, ordered bundle, or other cryptographic linkage that would make the operations inseparable once disclosed.
As a result, if a batch is assembled offchain and later fails or is otherwise revealed, the Guardian can still resubmit only a subset of those already-signed operations, or retry them in a different grouping, while remaining within the current authorization model. This is not an external bypass because only the Guardian can call the state-changing organization entrypoints, but it does mean atomic batch behavior is not enforced in a multi-layer fashion and ultimately depends on Guardian correctness.

Impact

Low. Batch atomicity is a trust assumption on the Guardian rather than a cryptographic guarantee, which could lead to partial resubmission of signed operations.

Recommendation

If batch-level atomicity is intended to be a hard security property, introduce a batch commitment into the signed payloads, such as a batch hash or batch identifier that every operation in the bundle must carry, and add an organization or wallet level batch entrypoint that enforces it onchain. Otherwise, document explicitly that preventing partial resubmission of a revealed batch is a trust assumption placed on the Guardian rather than a guarantee enforced by the contracts.

Developer Response

Batch-level atomicity is not intended to be a hard security property, and is intended to be the responsibility of the Guardian, rather than a guarantee enforced by the contracts.
We'll address this finding by explicitly documenting that preventing partial resubmission of a revealed batch is a trust assumption placed on the Guardian, and not a guarantee enforced by the contracts.
Implemented documentation changes explaining that batched atomicity is the responsibility of the guardian: - PR: #135 - Commit in `main`: 785eb14

2.8 Gas Savings Findings

2.8.1 Unnecessary Array and Struct type check in `_isParameterAllowedByConstraint()`

In `LibPolicyParameterConstraints`, the check for `ParamType.Array` or `ParamType.Struct` before the final `return false` is redundant and wastes gas.

Technical Details

The `_isParameterAllowedByConstraint()` function validates parameter constraints by checking each `ParamType` in sequence. The `ConstraintType.Any` wildcard is handled at the top of the function, so any Array or Struct parameter that reaches the type-specific checks necessarily has a non-Any constraint. Since Array and Struct only support the Any constraint, these will always return false:

```
if (pType == ParamType.Array || pType == ParamType.Struct) {
    return false;
}

// Case: The parameter is an unknown type
return false;
```

The explicit check for Array and Struct duplicates the behavior of the fallthrough `return false` immediately below it.

Impact

Gas Savings.

Recommendation

Remove the `if (pType == ParamType.Array || pType == ParamType.Struct)` block and let both cases fall through to the existing `return false`.

Developer Response

Implemented recommended gas savings change: - PR: #145 - Commit in `main`: 37d9a2f

2.8.2 Redundant `isGroup()` check before `isGroupMember()`

In `LibPolicyInitiator`, the `isGroup()` check before `isGroupMember()` is redundant because `isGroupMember()` already verifies that the given group ID corresponds to a current group in the organization.

Technical Details

The initiator validation for group-based policies performs two sequential checks:

```
if (!LibOrganizationGroups.isGroup(initiatorGroupId)) {
    return false;
}
return LibOrganizationGroups.isGroupMember(initiatorGroupId, initiatorAddress);
```

Since `isGroupMember()` internally validates that the group exists before checking membership, the preceding `isGroup()` call duplicates work and wastes gas.

Impact

Gas Savings.

Recommendation

Remove the standalone `isGroup()` check and rely on `isGroupMember()` to handle both group existence and membership verification.

Developer Response

Implemented recommended gas savings change and removed the redundant `isGroup` check in `LibPolicyInitiator`: - PR: #143 - Commit in `main`: eb653c5

2.8.3 Redundant validation checks in `OrganizationAccountFactoryBase.implementation()`

The `implementation()` function in `OrganizationAccountFactoryBase` validates that the account implementation address is nonzero and has code. These checks should already be enforced by `LibOrganizationAccountFactory.setAccountImplementation()` at the time the implementation is set.

Technical Details

`implementation()` performs two checks every time it is called:

```
if (impl == address(0)) {
    revert IOrganization.AccountImplementationNotSet();
}
if (impl.code.length == 0) {
    revert ERC1967Utils.ERC1967InvalidImplementation(impl);
}
```

If `setAccountImplementation()` already validates these conditions when the implementation address is stored, repeating them on every read adds unnecessary gas cost.

Impact

Gas Savings.

Recommendation

Remove the redundant checks from `implementation()` if `setAccountImplementation()` guarantees the stored address is always valid and nonzero.

Developer Response

Implemented recommended gas savings change and removed redundant validation checks in `OrganizationAccountFactoryBase.implementation()`: - PR: #148 - Commit in `main`: 8225e81

2.9 Informational Findings

2.9.1 Redundant chainId and organization fields in EIP-712 struct hashes

The EIP-712 typehashes defined in `LibOrganizationEIP712` redundantly include `chainId` and the organization address as struct members, even though both fields are already bound into the EIP-712 domain separator that wraps every struct hash.

Technical Details

`LibOrganizationEIP712.EIP712_DOMAIN_TYPEHASH` is defined as `EIP712Domain(string name,string version,uint256 chainId,address verifyingContract)`, and `getDomainSeparator()` populates `verifyingContract` with `address(this)` (the

organization) and `chainId` with `block.chainid`. Every signature is then wrapped by `computeTypedDataHash()`, which mixes the struct hash with this domain separator. The domain separator alone provides cross-chain and cross-contract replay protection.
Despite this, every operation typehash declared in `LibOrganizationEIP712` repeats `chainId` (and in most cases the organization address) as part of the struct schema. For example, in `_getAdminOperationHash()`:

```
bytes32 structHash = keccak256(
    abi.encode(
        LibOrganizationEIP712.ADMIN_OPERATION_TYPEHASH,
        uint8(operationType),
        keccak256(operationData),
        salt,
        expirationTimestamp,
        isApproval,
        block.chainid,
        address(this)
    )
);
```

The same pattern appears in `INITIATE_ACCOUNT_TRANSACTION_TYPEHASH`, `REVIEW_ACCOUNT_TRANSACTION_TYPEHASH`, `INITIATE_SIGNATURE_VALIDATION_TYPEHASH`, `REVIEW_SIGNATURE_VALIDATION_TYPEHASH` and `RECOVERY_SIGNATURE_VALIDATION_TYPEHASH`, all of which embed `chainId` and `organization` into the struct itself. These fields are redundant, slightly increase gas, and obscure which fields actually contribute to replay protection when reasoning about the schemas.

Impact

Informational.

Recommendation

Drop `chainId` and `organization` (or `verifyingContract`) from all typehashes in `LibOrganizationEIP712` and the corresponding `abi.encode` sites that build their struct hashes, relying solely on the EIP-712 domain separator for chain and verifying-contract binding.

Developer Response

Implemented recommended change: - PR: #147 - Commit in `main`: bd00886

2.9.2 Guardian signature validation accepts any enabled Safe module

The `_isValidGuardianSignature()` function in `LibOrganizationAccountSignature` accepts signatures from any module enabled on the Guardian Safe, not exclusively the intended `SafeExecutorModule`.

Technical Details

When validating a guardian signature, `_isValidGuardianSignature()` first checks whether the recovered signer matches the Guardian address directly. If not, it falls back to calling

`isModuleEnabled(address)` on the Guardian Safe to check if the recovered signer is any enabled module.

```
(bool success, bytes memory result) =
    guardianAddress.staticcall(abi.encodeWithSignature("isModuleEnabled(address)",
recoveredSignerAddress));
return success && result.length >= 32 && abi.decode(result, (bool));
```

This means any module enabled on the Guardian Safe can produce valid guardian signatures for the organization. If the Guardian Safe has additional modules enabled beyond the `SafeExecutorModule` (for example, a spending allowance module or a third-party automation module), those modules' authorized signers would also be accepted as valid guardian signatures. This expands the trust surface beyond the intended `AUTHORIZED_EXECUTOR` of the `SafeExecutorModule`.
Additionally, if an organization decides to switch its guardian to a different Safe, administrators must be aware that all enabled modules on the new Safe will automatically gain guardian signing authority.

Impact

Informational.

Recommendation

Document this behavior clearly so that organizations understand that any module enabled on the Guardian Safe can produce valid guardian signatures. This is especially important for organizations that decide to switch their guardian away from the Den infrastructure to a self-managed Safe, as they must ensure they fully understand and control which modules are enabled on the new Guardian Safe.

Developer Response

Acknowledged. We'll update documentation to document this behavior clearly, so organizations understand that any module enabled on the Guardian Safe can produce valid guardian signatures.
Implemented recommended documentation change: - PR: #133 - Commit in `main`: fc75958

2.9.3 BatchedTransaction does not validate well-formedness of packed payload

The `execute()` function in `BatchedTransaction` does not perform bounds checks on the packed transaction data, which means malformed or truncated payloads are silently parsed using zero-padded reads instead of being explicitly rejected.

Technical Details

The assembly loop in `execute()` uses `calldataload` and `calldatacopy` to parse the packed `[to (20)][dataLength (8)][data (N)]` format. The EVM zero-pads `calldataload` reads past the end of calldata and `calldatacopy` copies zeros for out-of-bounds regions. As a result, the function does not revert on truncated input.
Two types of malformed input are silently accepted:

1. If fewer than 28 bytes remain before the next header decode, the `to` address and `dataLength` fields are read from zero-padded calldata. This can produce phantom sub-transactions targeting `address(0)` with `dataLength=0`.
2. If the declared `dataLength` exceeds the remaining calldata, `calldatacopy` fills in zeros for the missing bytes, resulting in a call with partially correct, partially zeroed calldata.

In practice, most phantom or garbage calls will revert at the target (no matching selector, no fallback), causing the entire batch to revert atomically. However, target contracts that implement a `fallback()` function would accept such malformed calls. Since the `AUTHORIZED_EXECUTOR` controls the payload construction, the impact is limited to defense-in-depth against off-chain encoding bugs.

Impact

Informational.

Recommendation

Add explicit bounds checks in the assembly loop to fail early on malformed input:

```
// Before header decode
if lt(sub(endOffset, currentOffset), 28) {
    // revert MalformedBatch()
}

// After parsing dataLength, before calldatacopy
if lt(sub(endOffset, currentOffset), dataLength) {
    // revert MalformedBatch()
}
```

Developer Response

We will leave functionality as is and will not implement the recommended change. Only the guardian is intended to use this contract for batched transactions, and it is the responsibility of the guardian to encode data correctly. This behavior is similar to other "batched transaction" or "multisend" contracts that are commonly used in production today, where the user of the contract is expected to correctly encode data without contract-enforced checks.

2.9.4 `SafeExecutorModule` does not restrict call targets to known organizations

The `SafeExecutorModule` only prevents calls to the Safe itself, but does not verify that the target address is a known organization. This allows the authorized executor to call any arbitrary contract through the Safe.

Technical Details

In `executeOnBehalf()`, the module checks that the target is not the Safe address to prevent ownership and module modifications, but it does not validate that the target is a registered organization:

```
if (to == SAFE) {
    revert CannotCallSafe(to);
}
```

The authorized executor can call any contract address, including other modules or entities that may have control over the Safe through alternative paths. The `OrganizationFactory` could maintain a registry of deployed organizations that could be used to verify that the target is a legitimate organization.
Note that the same restriction would need to be replicated in the `BatchedTransaction` implementation for consistency.

Impact

Informational.

Recommendation

Consider adding a check that validates the target address is a registered organization deployed through the `OrganizationFactory`. This would provide an additional layer of defense by ensuring the Safe can only interact with known organizations.

Developer Response

We will leave this functionality as is and not implement the recommended change. The Guardian Safe will never be granted permission to perform privileged actions on any contracts besides the `OrganizationFactory` and MLS organization contracts, so restricting calls to only those contracts would not reduce or further limit the privileged calls the authorized executor can make.

2.9.5 `OrganizationPolicyBase.setPolicies()` relies on off-chain layers to ensure `newPoliciesRoot` matches the policy dump

Technical Details

`OrganizationPolicyBase.setPolicies()` commits administrators to a 32-byte `newPoliciesRoot` and a hash of the IPFS CID, but the contract has no way to verify that:

1. `newPoliciesRoot` is the Merkle root of any specific dump.
2. The IPFS payload at `ipfsCid` resolves to bytes whose Merkle root equals `newPoliciesRoot`.
3. Administrators reviewed the policies that `newPoliciesRoot` actually commits to.

This is intrinsic to the design: the IPFS CID is hashed using the IPFS multihash scheme over raw bytes; `newPoliciesRoot` is the keccak256 Merkle root of the parsed `(policyId, Policy)` leaves. The two cannot be cross-checked on-chain, and the contract cannot fetch IPFS content. NatSpec at `IOrganizationPolicy.sol:22-24` already declares the CID purpose as "for disaster recovery", consistent with the CID not being a signing-time trust primitive.
The `setPolicies` flow consequently depends on the off-chain layers of the project's three-layer security model to perform the consistency verification. If layers 1 and 2 perform their

independent verifications correctly, the realistic attacks against `setPolicies` (an admin proposing `(R_A, CID_of_B)` where `merkleRoot(B) ≠ R_A` to inject a hidden permissive policy set) fail. The on-chain layer is intentionally not the line of defense for this consistency check. The gap this finding identifies is documentation, not code.

Impact

Informational.

Recommendation

Add to docs a section that:

1. **Specifies the canonical dump format**. Exact serialization (ABI vs JSON), field ordering, leaf ordering when constructing the tree from the dump, and tie-breaking rules (e.g., sort by `policyId` ascending). The format must be deterministic so any compliant implementation produces the same `merkleRoot`.
2. **States the off-chain verification responsibilities explicitly**:

- The signing client MUST recompute `merkleRoot(dump) == newPoliciesRoot` from a dump the admin can review before presenting the EIP-712 payload for signature.
- The Guardian service MUST repeat the same verification independently before relaying on-chain.
- The CID is a disaster-recovery anchor, not a signing-time trust primitive; admins SHOULD NOT use IPFS-fetched content as the primary review source unless they also verify the recomputed root.

3. **Calls out that on-chain enforcement is intentionally absent for this check** so readers do not mistake the absence for a bug.
4. **Provides or links to a reference implementation** (open-source CLI / library) that any party can use to compute `merkleRoot(dump)` from a canonical dump. Reproducible builds preferred. Without this, the canonical-format specification is only paper, and admins are still forced to trust a closed-binary client.

Developer Response

Implemented documentation changes to explicitly mention off-chain verification responsibilities and call out that on-chain enforcement is intentionally absent for this check: - PR: #136 - Commit: 9174e60
The canonical dump format and a reference implementation for computing `merkleRoot(dump)` will be provided in a separate repository.

2.9.6 Pending signatures are not bound to a policy root or policy version

Technical Details

Policy updates replace the organization-wide Merkle root in `OrganizationPolicyBase.setPolicies()` / `LibOrganizationPolicy.setPolicies()`. However,

the EIP-712 digests used for transaction and ERC-1271 approvals do not bind signers to that root, to the policy leaf hash, or to any policy version/epoch.
For account transactions, the initiator and reviewer digests include `policyId` but nothing derived from `proofs.policy` or `policiesRoot`; see `LibOrganizationAccountTransaction._computeInitiatorHashFromParams()` and `_computeReviewHashFromParams()`.
For ERC-1271, the same pattern exists in `LibOrganizationAccountSignature._getInitiatorSignatureHash()` and `_getReviewSignatureHash()`. During validation, the contract then checks `proofs.policy` against the current Merkle root in `_validatePolicyBasedSignature()` and `LibOrganizationPolicy.isPolicyInOrg()`.
If a new root reuses the same `policyId`, already-collected signatures remain cryptographically valid and are reinterpreted under the new policy definition. The result could be a semantic mismatch between what signers approved offchain and what the contract enforces onchain at execution time.
Neither the NatSpec of `setPolicies()` / `LibOrganizationPolicy.setPolicies()` nor any file under `docs/` currently states that updating `policiesRoot` does not invalidate outstanding approvals and that reusing a `policyId` preserves pending signatures until they expire (the existing notes on signature invalidation in `docs/SIGNATURES.md` cover only nonce burning on execute/reject).

Impact

Informational.

Recommendation

Document this behavior clearly.

Developer Response

Implemented recommended documentation changes: - PR: #132 - Commit in `main`: 7ef53fb

2.9.7 Rate limits and amount thresholds are unreliable for anyToken policies

Rate limits and per-transaction amount thresholds operate on raw token amounts without accounting for token identity or decimal precision. When combined with `anyToken = true` policies, these controls become meaningless because they mix values from tokens with different decimals and economic worth into a single comparison or usage bucket.

Technical Details

Token transfer policies can set `anyToken = true` to authorize transfers of any token. Amount controls in `LibPolicyTokenTransfer._isTokenAmountAllowedByPolicy()` compare the raw extracted amount against a configured threshold, while `LibPolicyRateLimits` tracks cumulative usage per time window using the same raw amounts. The rate limit usage key computed by `computeUsageKey()` does not include the token contract address, so all tokens share the same bucket.

This creates two problems when multiple tokens with different decimals are used under the same policy:

1. A threshold calibrated for an 18-decimal token (e.g., `1e18` for ”1 token”) would permit `1e18` raw units of a 6-decimal token like USDC, which represents `1e12` USDC in economic terms.
2. Time-interval rate limits accumulate raw amounts across different tokens in the same bucket, making the cumulative cap arbitrary when tokens have different decimal scales or economic values.

Impact

Informational.

Recommendation

Avoid combining `anyToken` policies with amount thresholds or time-interval rate limits. If multi-token spending caps are needed, use explicit per-token policies with thresholds calibrated to each token’s decimals, or include the token contract address in the rate limit usage key and maintain per-token buckets.

Developer Response

We now revert any transaction whose policy combines `anyToken` with an amount threshold or a time-interval rate limit, eliminating this class of misconfiguration: - PR: #151 - Commit in `main`: 2f748bf

2.9.8 Dual initialization checks may cause confusion

Multiple contracts use custom `isInitialized()` functions based on domain-specific state, alongside OpenZeppelin’s `Initializable` library which tracks initialization independently. Having two separate mechanisms could lead to confusion about the contract’s state.

Technical Details

In `LibOrganizationInitialization`, the `initialize()` function checks `isInitialized()` (which returns `adminCount > 0`) before proceeding. The organization also inherits from OpenZeppelin’s `Initializable`, which provides its own `_initialized` flag and `initializer` modifier. These two guards operate independently: one is domain-specific (admin count) and the other is a generic reentrancy-style guard from the upgradeable proxy pattern.
A similar pattern exists in `ImplementationWhitelistImplementation`, where `isInitialized()` returns `owner() != address(0)`. This relies on the assumption that the owner is always set during initialization, but would incorrectly report the contract as uninitialized if the owner ever renounces ownership via `renounceOwnership()`, even though the contract was properly initialized.

Impact

Informational.

Recommendation

Document the relationship between the two initialization mechanisms and clarify which one serves as the authoritative check. Alternatively, consolidate them to use a single source of truth. For the whitelist contract, consider using OpenZeppelin's initialization state directly instead of relying on ownership as a proxy for initialization status.

Developer Response

Consolidated both `isInitialized()` implementations to read OpenZeppelin's initialization state directly, establishing a single source of truth rather than documenting the relationship between the two previous initialization mechanisms: - PR: #149 - Commit in `main`: 988109e

2.9.9 `getActualDestination()` implementation could be simplified

The `getActualDestination()` function in `LibPolicyDestination` uses two sequential checks (empty calldata, then token transfer detection) where the logic could be expressed more directly as a single ternary condition.

Technical Details

The current implementation first checks for empty calldata (native ETH transfer), then checks whether the transaction is a token transfer:

```
// Case: The transaction is a native token transfer
if (data.length == 0) return to;

// Case: The transaction is a contract interaction
if (!TokenTransferUtils.isTransactionTokenTransfer(data, value)) return to;

// Case: The transaction is an ERC-20 token transfer
// Extract the recipient address from the transfer function call
return TokenTransferUtils.extractERC20TransferRecipient(data);
```

Since `isTransactionTokenTransfer()` already handles both native and ERC-20 transfers, this could be simplified to a single check that returns the ERC-20 recipient only when the transaction is a token transfer, and `to` otherwise. The current version is functionally correct but adds an unnecessary branching step.

Impact

Informational.

Recommendation

Consider simplifying the function to a single expression for readability.

```
return isTransactionERC20TokenTransfer(data, value) ? TokenTransferUtils.
extractERC20TransferRecipient(data) : to;
```

Developer Response

Implemented recommended change and simplified `getActionDestination` to the recommended ternary: - PR: #146 - Commit in `main`: ed87f47

2.9.10 Cross-chain organization address stability depends on initial implementation address

The `OrganizationFactory` embeds the implementation address in the `OrganizationProxy` creation bytecode, which is used as the CREATE2 salt input. This creates a dependency between cross-chain address determinism and the initial implementation address.

Technical Details

`_getOrganizationProxyBytecode()` encodes the implementation address into the proxy's constructor arguments:

```
return abi.encodePacked(type(OrganizationProxy).creationCode, abi.encode(
implementationAddress, whitelistAddress));
```

Because CREATE2 addresses are derived from the deployer, salt, and init code hash, the organization's address is tied to the specific implementation address used at deployment time. This has two implications:

1. When the implementation is upgraded on a new chain, new organizations must first be deployed with the original implementation address and then upgraded, in order to produce the same address.
2. The original implementation address cannot be removed from the whitelist, as it must remain deployable to preserve address consistency across chains.

Impact

Informational.

Recommendation

Document these constraints explicitly for operators managing cross-chain deployments. Alternatively, consider removing the implementation address from the proxy's constructor arguments and setting it post-deployment, so that the CREATE2 address does not depend on the implementation version.

Developer Response

Implemented the alternative recommended change and removed the implementation address from the `OrganizationProxy` constructor and made it so that the proxy's implementation address is set post-deployment, so that CREATE2 addresses do not depend on the implementation version: - PR: #152 - Commit in `main`: 00f3079

yAudit Response

The auditors note that the added `setInitialImplementation(address)` function in the proxy contract may produce a selector clash with a function of the same selector in the implementation contract. The development team has acknowledged and accepted this risk.

2.9.11 Dirty upper bytes not cleared in `_extractContractInnerSignature()`

The `_extractContractInnerSignature()` function in `SignatureUtils` copies signature bytes using 32-byte `mstore` operations but does not clear potential dirty bytes in the last word when `sigLength` is not a multiple of 32. This is inconsistent with the approach used in `BytesUtils.sliceRange()`.

Technical Details

The assembly loop in `_extractContractInnerSignature()` copies data in 32-byte chunks. When `sigLength` is not aligned to 32 bytes, the final `mstore` writes a full 32-byte word, potentially leaving dirty bytes beyond the declared length of the `contractSig` bytes array. While Solidity's high-level operations respect the length field and ignore trailing bytes, low-level consumers or hashing operations that read past the declared length could observe inconsistent data.

Impact

Informational.

Recommendation

Clear the trailing bytes after the copy loop, similar to how `BytesUtils.sliceRange()` handles partial words, or document the assumption that consumers always respect the length prefix.

Developer Response

We will leave `_extractContractInnerSignature` as it is and not implement the recommended change. There is no path in which a valid signature fails validation, an invalid signature is accepted, or any data is exfiltrated or injected: every consumer of the returned bytes respects the declared length prefix. Replacing `_extractContractInnerSignature` with `BytesUtils::sliceRange()`, or adding equivalent masking for trailing bytes, would push `OrganizationImplementation` past the bytecode size limit for deployment and would require larger changes to reduce the overall bytecode size.

2.9.12 Missing external getters for internal state queries

Several internal state queries tracked by the organization have no corresponding external getter, preventing off-chain integrations and other contracts from accessing this information without relying on events.

Technical Details

1. **Deployed accounts**: The `deployedAccounts` mapping in `LibOrganizationAccountFactoryStorage` tracks which accounts were deployed by the organization, and `LibOrganizationAccountFactory.isAccountDeployedByOrganization()` exists as an internal library function, but no public wrapper is exposed in `OrganizationAccountFactoryBase`.
2. **Deleted groups**: The group system tracks deleted groups internally, but `OrganizationGroupsBase` only exposes `isGroup()` (which returns false for deleted groups) and `isGroupMember()`. There is no way to distinguish between a group that was never created and one that was created and later deleted.

Impact

Informational.

Recommendation

Expose `isAccountDeployedByOrganization()` as a public view function in `OrganizationAccountFactoryBase`, and add an `isDeletedGroup()` or `getGroupStatus()` view function in `OrganizationGroupsBase`.

Developer Response

We will leave the external getters as they are and not implement the recommended changes. These additional getters are not required for MLS to function, and adding them would push `OrganizationImplementation` past the bytecode size limit for deployment, which would require larger changes to reduce the overall bytecode size.

2.9.13 Unused `AccountTransactionRejection` enum value

The `OperationType` enum in `CommonTypes.sol` defines an `AccountTransactionRejection` value that is never used anywhere in the codebase. Unused enum values can lead to confusion about intended functionality.

Technical Details

The `AccountTransactionRejection` value is the last entry in the `OperationType` enum. No function or library references this value, and no signature digest includes it. Its presence suggests a feature that was removed or never implemented.

Impact

Informational.

Recommendation

Remove `AccountTransactionRejection` from the `OperationType` enum to avoid confusion.

Developer Response

Implemented suggested change and removed unused `AccountTransactionRejection` enum value from `OperationType`: - PR: #139 - Commit in `main`: e179625

2.9.14 Add `onlyProxy` modifier to `OrganizationImplementation.upgradeToAndCallWithAuthorization()`

Technical Details

`OrganizationImplementation.upgradeToAndCallWithAuthorization()` is the custom upgrade entrypoint. It is protected by `onlyGuardian`, validates admin authorization, performs whitelist checks, and then forwards into the inherited UUPS `upgradeToAndCall`.
Unlike the inherited UUPS entrypoint, it does not explicitly enforce proxy context with `onlyProxy`.
Today this is still fail-closed in practice:

- A direct call on the implementation reaches `onlyGuardian`.
- `onlyGuardian` reads the guardian slot from the implementation's own storage.
- The implementation constructor calls `_disableInitializers()`, so the guardian slot remains unset.
- As a result, direct implementation calls revert before the upgrade flow can proceed.

So the current protection works, but it is state-dependent rather than structural. The proxy-only invariant for this entrypoint currently relies on implementation storage remaining uninitialized, instead of being enforced directly by the function itself.

Impact

Informational.

Recommendation

Add `onlyProxy` to the entrypoint before `onlyGuardian` so the execution-context check fails first:

```
function upgradeToAndCallWithAuthorization(
    address newImplementation,
    bytes calldata data,
    AdminAuthParams calldata authParams
) external override onlyProxy onlyGuardian {
    // body unchanged
}
```

Developer Response

Fixed. Implemented recommended change (adding onlyProxy before onlyGuardian):

- PR: #150
- Commit in main: 13cd351

2.9.15 `SafeExecutorModule` documents an EOA executor but does not enforce it onchain

Technical Details

The Safe module documentation and deployment flow consistently describe `AUTHORIZED_EXECUTOR` as an EOA. See `GUARDIAN_PROTECTION.md`, `DEPLOYMENT.md`, and `DeployGuardianSafeModule.s.sol`.
Onchain, though, `SafeExecutorModule` only checks that `authorizedExecutor != address(0)`. It does not enforce `authorizedExecutor.code.length == 0` or otherwise constrain the signer type.
That means the module can be deployed with a contract as `AUTHORIZED_EXECUTOR`. In that configuration:

- `executeOnBehalf()` authorizes calls based only on `msg.sender == AUTHORIZED_EXECUTOR`, regardless of whether that sender is an EOA or contract.
- `isValidSignature()` uses `SignatureUtils.tryRecoverSigner()`, which supports both EOA and ERC-1271 signers, so the module's signing semantics can effectively be delegated to that configured contract address.

Impact

Informational.

Recommendation

Make the design intent explicit in code:

- If the executor must be an EOA, reject contract addresses in the constructor.
- If contract executors are intentionally allowed, update the contract comments and deployment docs to describe the broader signer model accurately.

Developer Response

Implemented changes to documentation and comments to make it clear that both EOAs and smart contracts are allowed and supported as the authorized executor: - PR: #137 - Commit in `main`: d8dc90d

2.9.16 Strengthen whitelist proxy initialization

Technical Details

`ImplementationWhitelistProxy` only rejects `initData.length == 0`, but it accepts any other payload. During construction, OpenZeppelin's `ERC1967Proxy` forwards any non-empty `_data` to `ERC1967Utils.upgradeToAndCall()`, which only requires the delegatecall to succeed. That means `initData` does not need to call `initialize(...)`. Any non-reverting selector exposed by `ImplementationWhitelistImplementation`, such as `isInitialized()` or `isImplementationWhitelisted()`, can return successfully while leaving `owner == address(0)` and the initializer still callable at `initialize(...)`.

Impact

Informational.

Recommendation

Strengthen the `ImplementationWhitelistProxy` constructor:

```
constructor(
    address implementation,
    address initialOwner,
    address[] memory organizationImplementations,
    address[] memory accountImplementations
)
    ERC1967Proxy(
        implementation,
        abi.encodeCall(
            IImplementationWhitelist.initialize,
            (initialOwner, organizationImplementations, accountImplementations)
        )
    )
{}
```

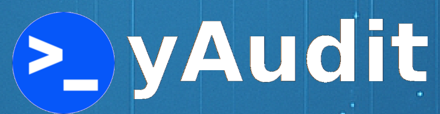
Developer Response

We will leave the constructor as is and not implement the recommended change. This `ImplementationWhitelistProxy` is only ever deployed once per chain by our team, and it is deployed via CREATE2, so its address is a deterministic function of the init code — including the encoded `initData` passed to `initialize`. If `initData` were ever encoded incorrectly, the proxy would deploy to a different address than the expected canonical one. We would therefore detect the error immediately, the correct canonical address would remain unaffected and undeployed, and we could simply redeploy with correctly encoded `initData` to obtain the correct contract at the correct address.

2.10 Final Remarks

The codebase is mature, well structured, and reflects a deliberate three-layer security model where the signing client, the Guardian service, and the onchain contracts each enforce policy independently. Several findings sit at the boundary between offchain policy construction and

onchain enforcement. The contracts verify the submitted signatures, proofs, roots, and constraints, while the client, Guardian, and operational process add further enforcement of policy intent and batching assumptions on top of those guarantees. Overall the contracts are well-structured, and the offchain signing-client and Guardian layers reinforce the guarantees enforced onchain, together providing defense in depth across the full stack.



Den MLS Wallet

Completed 2026-04-29