OpenZeppelin | security

OpenZeppelin Contracts Release v5.1 Audit

OpenZeppelin

October 3, 2024

Table of Contents

Summary

Type	Library
Phase 1	From 2024-07-29 To 2024-08-14
Phase 2	From 2024-08-19 To 2024-09-06
Languages	Solidity

Timeline

Total Issues	42 (31 resolved)
Critical Severity Issues	0 (0 resolved)
High Severity Issues	0 (0 resolved)
Medium Severity Issues	3 (2 resolved)
Low Severity Issues	9 (8 resolved)
Notes & Additional Information	28 (19 resolved)
Client Reported Issues	2 (2 resolved)

Scope

We audited the OpenZeppelin/openzeppelin-contracts repository at commit aba9ff6. The audit consisted of two phases.

Phase 1

The scope of this audit was limited to the changes introduced in some contracts that were previously audited for the v5.0.0 release.

In scope were the following files:

```
contracts
├── access
│   ├── extensions
│   │   └── AccessControlEnumerable.sol
│   └── manager
│       └── AccessManager.sol
├── governance
│   ├── Governor.sol
│   ├── extensions
│   │   └── GovernorCountingSimple.sol
│   └── utils
│       └── Votes.sol
├── metatx
│     └── ERC2771Forwarder.sol
├── proxy
│   ├── Clones.sol
│   ├── ERC1967
│   │   └── ERC1967Utils.sol
│   └── transparent
│       └── TransparentUpgradeableProxy.sol
├── token
│   ├── ERC1155
│   │   ├── ERC1155.sol
│   │   └── extensions
│   │       └── ERC1155Supply.sol
│   ├── ERC20
│   │   └── utils
│   │       └── SafeERC20.sol
│   ├── ERC721
│   │   ├── ERC721.sol
│   │   └── extensions
│   │       └── ERC721Enumerable.sol
│   └── common
│       └── ERC2981.sol
```

```
└── utils
    ├── Address.sol
    ├── Arrays.sol
    ├── Base64.sol
    ├── Create2.sol
    ├── StorageSlot.sol
    ├── Strings.sol
    ├── cryptography
    │   ├── MerkleProof.sol
    │   └── SignatureChecker.sol
    ├── math
    │   ├── Math.sol
    │   ├── SafeCast.sol
    │   └── SignedMath.sol
    └── structs
        ├── Checkpoints.sol
        ├── DoubleEndedQueue.sol
        └── EnumerableMap.sol
```

Phase 2

In scope were the following files:

```
contracts
├── finance
│   └── VestingWalletCliff.sol
├── governance
│   └── extensions
│       └── GovernorCountingFractional.sol
├── token
│   ├── ERC20
│   │   └── extensions
│   │       ├── draft-ERC20TemporaryApproval.sol
│   │       └── ERC1363.sol
│   ├── ERC721
│   │   └── utils
│   │       └── ERC721Utils.sol
│   └── ERC1155
│       └── utils
│           └── ERC1155Utils.sol
└── utils
    ├── Errors.sol
    ├── Packing.sol
    ├── Panic.sol
    ├── ReentrancyGuardTransient.sol
    ├── SlotDerivation.sol
    ├── cryptography
    │   ├── Hashes.sol
    │   ├── P256.sol
    │   └── RSA.sol
    └── structs
        ├── CircularBuffer.sol
```

```
      ├── Heap.sol
      └── MerkleTree.sol
```

Overview - Phase 1

Version 5.1 of the OpenZeppelin Contracts library introduces minor changes to previously existing contracts. The following modifications were made across the majority of the in-scope contracts:

- **Improved documentation**: Many contracts have had their docstrings reworked. They have either been improved or have been adapted to describe the new functionalities.
- **Improved handling of common errors**: Common errors have now been isolated into a separate `Panic` or `Errors` library.
- **Improved return parameters**: Many functions have been changed to have their return parameters named.

For a complete list of specific changes made in individual files, one can refer to the changelog from version 5.0.0 to the latest one.

During the course of the audit, some behaviors were observed in the system that were considered worth mentioning to the community.

- **`AccessManager.execute` Reverts on Failed External Calls**

  The `AccessManager` contract provides a mechanism for role-based access control, allowing certain operations to be scheduled and executed with specified delays. This system ensures that only authorized users can perform certain actions. Once a call is `scheduled`, depending on the role, the `caller` needs to wait until the end of an execution delay period before they can `execute` the scheduled call. If a scheduled call is malicious, an address with the `admin` or `guardian` role can cancel the schedule before it is executed.

  Once the delay period has passed (if it is applicable for a given role and caller), the `execute` function within this contract is responsible for carrying out these operations. It achieves this by invoking the target contract's function through a low-level call, with the option to send value along with the call.

  The `execute` function utilizes the `Address.functionCallWithValue` function to perform the external call to the target contract. This function reverts if the external call fails. While this ensures that failed operations do not proceed, it also means that if the external call fails, the operation is not marked as executed. This allows the caller to retry the same operation without needing to reschedule it, as long as the operation has not expired. In certain situations, depending on the `target` and the reason of the failed execution, retrying the execution can result unsuccessful attempts thereby causing wastage of gas.

Overview - Phase 2

Version 5.1 of the OpenZeppelin Contracts library introduces several new features, including the following:

Adding Cliff to the Vesting Wallet

To address the *Lack of Inclusion of a "Vesting Cliff" Feature* issue from the audit of the `v5.0.0` release, the Contracts team has created a `VestingWalletCliff` contract, which adds a cliff period to the vesting schedule.

Support for Fractional Voting

The `GovernorCountingFractional` contract allows delegates to decide how they want to use their voting power. The delegates can either fractionally split their voting weight into `Against`, `For`, and `Abstain` votes (called *fractional voting*), or cast their entire voting weight into one of the three options (called *nominal voting*).

To identify if the voting is nominal or fraction, the `_countVote` function of the `GovernorCountingFractional` contract utilizes the `support` and `bytes memory params` parameters in the function signature. If `support` is `255` and `params` has exactly 48 bytes, the vote is considered fractional.

While traditional voting mechanisms allow an account to participate in a voting process only once, the `GovernorCountingFractional` contract allows for rolling voting, whereby one account can vote for a proposal with a portion of its total weight and then subsequently vote again with the remaining portion of its weight.

Given that fractional voting restricts the number of casted votes (in each category) to 128 bits, depending on the decimals of the underlying token, a voter may have to split their vote into multiple vote operations. This has been properly documented in the contract.

Support for ERC-1363 and ERC-7674

The `ERC1363` contract extends the functionality of ERC-20 tokens by atomically allowing code execution on the target contract after a transfer or approval. The `transferAndCall` and `transferFromAndCall` functions call the `_checkOnTransferReceived` hook which calls `IERC1363Receiver-onTransferReceived` on the recipient address. Similarly, the `approveAndCall` functions call the `_checkOnApprovalReceived` hook which calls `IERC1363Spender-onApprovalReceived` on the spender address.

The `draft-ERC20TemporaryApproval` contract implements the ERC-7674 standard which introduces temporary approval extensions for ERC-20 tokens. By using transient storage, the `owner` can approve an allowance for the `spender` which is only valid for the current transaction.

Code Refactor

In the v5.1 release, the `checkOnERC721Received` function which is used to verify if a recipient contract implements the `IERC721Receiver-onERC721Received` hook, has been moved to a new `ERC721Utils` library. Similarly, the `checkOnERC1155Received` and `checkOnERC1155BatchReceived` functions have been moved to the `ERC1155Utils` library.

Addition of New Utilities

A set of new contracts has been added to the `utils` suite. These contracts provide implementations for cryptographic libraries such as `RSA` and `P256`, and new data structures such as `CircularBuffer`, `Heap`, and `MerkleTree`. The `ReentrancyGuardTransient` library implements the `ReentrancyGuard` using transient storage.

A comprehensive list of changes made in the v5.1 release can be found in the changelog.

Security Model and Trust Assumptions

Auditing libraries requires a shift in focus due to their composability within blockchain protocols. While the scope of an audit is typically limited to the code itself, this expands when it comes to libraries because of their potential internal and external integrations. Libraries act as foundational components for many protocols. This means that their security is influenced not just by their internal robustness, but also by how they are utilized by integrators. Therefore, ensuring a library's security involves not only reviewing the code but also anticipating its various use cases and integration scenarios.

In addition, the complexity grows because while a library must cover a wide range of potential use cases, the responsibility for secure implementation often lies with the developers who integrate it into their projects. A library's security risks can multiply depending on how well developers understand and utilize its contracts. This necessitates extra care to ensure that all potential threats, both direct and indirect, are either identified and addressed, or documented so that the developers are aware of the security risks.

Medium Severity

M-01 `AccessManager`'s `schedule` Function Misses `minGas` and `minValue` - Phase 1

The `schedule` function of the `AccessManager` contract allows users to schedule some calls to `target`. This operation generates an `operationId` by hashing the provided `caller` address, `target` address, and `data`.

However, when calling `execute` on a scheduled operation, the call is performed using `msg.value` as the attached value. This `msg.value` can be anything and is not meant to be a part of `operationId`. As a result, there can be different execution outcomes for the same `operationId` if the `target` contract implements a logic that depends on `msg.value`. To some extent, the same argument can be made for the provided gas in the `execute` function, given that the `target` might have custom logic based on that as well.

Consider adding minimum committed `minValue` and `minGas` parameters to the `schedule` function and check those in the `execute` function implementation. Alternatively, consider making them exact values instead of minimal ones.

Update: *Acknowledged, not resolved. The OpenZeppelin Contracts team stated:*

> *Both gas and value were left out of the `operationId` considering that the operation can only be executed by its original scheduler. For this reason, the proposer has full control of how the proposal is executed and can evaluate the execution requirements of the target at that moment.*
>
> *The team considers that specifying exact `gas` and `value` allows to DoS an operation by manipulating the target requirements. Similarly, specifying minimum values does not eliminate the possibility of different execution outcomes.*
>
> *For these reasons, and considering that this change would be breaking, we have decided not to include `gas` and `value` parameters as part of an operation.*

M-02 Proving Empty Set Inclusion May Lead to Issues for Integrators in `MerkleProof.multiProofVerify` - Phase 1

The `multiProofVerify` function of the `MerkleProof` library allows for proving the inclusion of an empty set of leaves in the root of a tree. While this logic may be valid in isolation, it poses a significant risk during integration into other codebases. Particularly when the integrator accepts arrays of arguments from users and relies on this function to validate the inclusion of a leaf in the tree.

The issue occurs when the `proof` array only contains the Merkle root, and both the `proofFlags` and `leaves` arrays are empty. Despite the absence of a complete proof, the function still computes the Merkle root and considers the proof as valid. This can lead to a scenario where a user can bypass proper validation by submitting an incomplete proof that the function accepts as valid and returns `true`. A secret gist has been created with PoC demonstrating the potential problem.

To ensure the safety of integrators, consider adding an additional argument such as `treeHeight` to the function and validate that the arguments provided are indeed for that height. This would prevent situations where, for example, an integrator assumes a tree height of 5 but the user can pass a set of arguments corresponding to the proof of an empty set at the root and the function still returns `true`. Alternatively, consider adding a warning for integrators which could help mitigate this risk.

Update: *Resolved in pull request #5144 at commit c304b67 and in pull request #5142 at commit bcd4beb. The OpenZeppelin Contracts team stated:*

> *The empty set validity is a property we expect in the `multiProofVerify` function according to a consistent no-op policy that we have followed throughout the library. As such, we consider this inclusion proof a complete one and acknowledge the risks of using the `multiProofVerify` function without validating the content of its leaves. However, we did not find any meaningful use case where the leaves are used only for proof validation without further validation or usage. Considering this, we think the `treeHeight` argument might increase the algorithm's complexity and undermine developer experience while not preventing a concrete impact. We added a note to make the validity of the empty set proof explicit but decided against changing the function semantics to preserve backward compatibility.*

M-03 Dirty Bytes Can Manipulate Derived Mapping Slot - Phase 2

The `SlotDerivation` library allows developers to derive a value slot given the slot of a mapping and a key. There are several functions depending on various key types. Two of these supported key types are `address` and `bool`. These two types allocate 20 bytes and 1 byte, respectively. However, if these input keys have previously been manipulated or assigned with assembly, it is possible that the upper bytes are dirty (non-zero). These dirty bytes would lead to a different hash and therefore different storage location. The impact of this flaw is very context-dependent with the caveat of assembly manipulation, but can escalate to severe issues.

Consider cleaning the upper bytes of the `address` and `bool` type key before hashing these values.

Update: *Resolved in pull request #5195 at commit 9f0960d.*

Low Severity

L-01 Royalty Calculation May Result in Zero Token Transfers Between Buyer and Seller of NFT - Phase 1

In `ERC2981.sol`, the `_setTokenRoyalty` function checks and reverts if `feeNumerator` is greater than `_feeDenominator()` to ensure that `royaltyAmount` is always less than the `salePrice`. However, the function accepts `feeNumerator` to be equal to `_feeDenominator()`. In the `royaltyInfo` function, if `feeNumerator` is equal to `_feeDenominator()`, the `royaltyAmount` becomes equal to `salePrice`, which could translate to the total price of the NFT sale being paid as royalty and no transfer of ERC-20 tokens between the buyer and the seller.

It is essential to note that most NFT marketplaces that support royalty payments have distinct addresses, one for transfer of sale price, and another for paying royalty. In general, the royalty is paid to the creator of the NFT and the sale price is transferred from the buyer to the seller (or owner) of the NFT. The ERC states that royalty should be a percentage of the sale price. Therefore, the royalty being equal to 100% of the sale price is a valid scenario. However, a transfer of zero amount of tokens between the buyer and the seller of the NFT could be incompatible with protocols that use ERC-20 tokens which do not allow zero-token transfers.

Consider documenting the aforementioned behavior so that the protocols integrating with this contract are aware of potential zero-amount transfers.

Update: *Resolved in pull request #5173. The OpenZeppelin Contracts team stated:*

> *The team agreed with the risks of paying the `royaltyAmount` using an ERC-20 token that reverts on 0-value transfers. We are documenting this issue with a note for integrators to consider.*

L-02 Missing Docstrings - Phase 1

Throughout the codebase, multiple instances of missing docstrings were identified:

- The `ADMIN_ROLE` state variable in `AccessManager.sol`
- The `PUBLIC_ROLE` state variable in `AccessManager.sol`
- The `upgradeToAndCall` function in `TransparentUpgradeableProxy.sol`

Consider thoroughly documenting all functions (and their parameters) that are part of any contract's public API. Functions implementing sensitive functionality, even if not public, should be clearly documented as well. When writing docstrings, consider following the Ethereum Natural Specification Format (NatSpec).

Update: *Resolved in pull request #5168.*

L-03 Custom Functions Might Modify Memory - Phase 1

The custom `hasher` function of the `MerkleProof` library and the `comp` comparator function of the `Array` library are arbitrary functions passed as input parameters by integrators. There are no restrictions on these functions apart from the list of input and output parameters. As such, integrators might maliciously or accidentally code these functions in a way that modifies the memory state whenever they are executed. Depending on the logic of these functions, modifications done to memory can result in unexpected behavior.

While balancing the trade-offs between providing flexible library code and the risk of side effects like memory manipulation, consider adding this edge case as a warning in the documentation.

Update: *Resolved in pull request #5174. The OpenZeppelin Contracts team stated:*

> *We acknowledge the risks of custom hashing functions so we are documenting that memory side effects should be considered when using function pointers*

L-04 Incorrect or Misleading Docstrings - Phase 1

Throughout the codebase, multiple instances of incorrect or misleading docstrings were identified:

- This docstring in the `ERC1967Utils.sol` incorrectly mentions that the library is an "abstract contract".
- This docstrings within the `_encode` function in `Base64.sol` states that, if padding is absent, the `data.length` is rounded up and then multiplied by 4. However, in the code implementation, `data.length` is first multiplied by 4 and then rounded up.

Consider updating the aforementioned instances of misleading docstrings for improved code clarity and readability.

Update: *Resolved in pull request #5168.*

L-05 Different Pragma Directives Are Used - Phase 2

In order to clearly identify the Solidity version with which the contracts will be compiled, pragma directives should be consistent across file imports.

`MerkleTree.sol` has the pragma directive `pragma solidity ^0.8.0;` and imports the file `Panic.sol`, which has a different pragma directive `pragma solidity ^0.8.20;`. In addition, `Hashes.sol` is the only other file in the entire codebase that uses `pragma solidity ^0.8.0;` directive.

Consider using the same floating pragma version in all files.

Update: *Resolved in pull request #5198. The OpenZeppelin Contracts team stated:*

> *We are increasing the pragma version to 0.8.20. This is consistent with the rest of the library.*

L-06 Vesting Can Start Before Cliff Ends - Phase 2

A cliff is a specific period during which the tokens are locked and the holder cannot claim the allocated tokens. However, the `_vestingSchedule` function in the `VestingWalletCliff` contract allows the vesting to begin when the `timestamp` becomes equal to the `cliff()`.

Consider starting the vesting period after the cliff has ended.

Update: *Acknowledged, not resolved. The understanding of the cliff point in time is subjective and the OpenZeppelin Contracts team prefers that the cliff timestamp marks the beginning of the vesting. The OpenZeppelin Contracts team stated:*

> *The interpretation of the cliff from `VestingWallerCliff` matches the one we observed in other similar contracts, such as this one from ThirdWeb.*
>
> *In any case, this difference of interpretation (< vs <=) only changes the outcome for one particular second. This is insignificant over the common duration of cliffs (months) and vestings (years). For this difference to even be visible, you would need a block to be produced at the exact second the cliff ends (there is an 8% chance that this block even exists, considering 12s between blocks), and you would have to release the asset in that exact block. Even if that were to happen, re-submitting the same transaction in the next block would "resolve" things.*
>
> *Here we value lower gas cost (of < over <=) and, more importantly, consistency with existing vesting wallets.*

L-07 Over-Engineered `Heap` - Phase 2

A heap is a binary-tree-based data structure that satisfies the heap property, which is that the value of a parent node is always less than the values of its children. `Heap.sol` uses an array of `Node` objects to implement a heap, where each `Node` consists of a `value`, an `index`, and a `lookup`. Currently, the insertion of a new element is done by pushing a new node at the end of the array, comparing the value of this node to the value of the parent node, and swapping the indexes and lookups if the value of the new element is less than the value of the parent node. This process is repeated till the heap property is met.

The root element is removed by reading the root and the last elements of the heap. If the `rootNode` is not the last element of the array, the value of the root is replaced with the value of the last element, and the respective indexes and lookups are exchanged. Once copied to the root node, the last element is popped out of the array. The heap then rebalances (or *heapifies*) its nodes by comparing the root value to its children's values until the heap property is met, thereby swapping the indexes and lookups wherever necessary. Similarly, a replacement of the root element is done by replacing the value of the root element with the new value, and heapifying until the heap property is met.

The current implementations of the `insert`, `pop`, and `replace` functionalities are difficult to follow due to the necessary reading and writing of the `indexes` and `lookups`. Additionally, the `_swap` function along with the `_siftUp` or `_siftDown` functions costs more gas to swap the `indexes` and `lookups` of two nodes as compared to simply swapping the `values`.

To reduce the complexity of the codebase, consider simplifying the heap structure by using a linear `uint256` array that stores the values of each node. The array is created such that the root is always at index 0 (i.e., the first element of the array). For each node at index `i`, the value of the left child is found at index `i * 2 + 1` and the right child is at `i * 2 + 2`.

Update: *Resolved in pull request #5190 at commit 71fb803 and in pull request #5215 at commit 2843690. The OpenZeppelin Contracts team stated:*

> *As suggested, the Heap is updated to remove the dependency on `index` and `lookup` since it is redundant to store them. This way, we simplify the implementation and improve readability while keeping the core array-based implementation that will leverage cheaper adjacent storage accesses after the Verkle EVM upgrade.*

L-08 Small Public Exponents in RSA - Phase 2

The `pkcs1` function of `RSA.sol` allows small values of the public exponent e (e.g., $e = 3$). Small values of e have been known to be vulnerable to attacks such as Coppersmith's attack, which the `RSA` implementation seems to be vulnerable to.

In cases where the public key is small (e.g., $e = 3$), Coppersmith's attack on the RSA signature scheme allows an attacker to forge a signature S for an arbitrary message m without knowledge of the private key d. This is achieved by solving the following equation using the Coppersmith's method [1, 2]:

$$S^3 = M \bmod n$$

The above is equivalent to computing the third root of the padded message M modulo n, which is feasible for $e = 3$, when the structure of M is partially known as in the case of the PKCS#1 v1.5 padding scheme and the solution S is smaller than the modulus n. Specifically, Coppersmith's theorem states that for the attack to be successful, the length of S should be at most $n^{\frac{1}{e}-\epsilon}$ for $\frac{1}{d} > \epsilon > 0$, where e is the small public exponent (e.g., $e = 3$). Roughly, the length of S should not be larger than $n^{\frac{1}{3}}$ or ≈ 341 bits for n of around 1024 bits.

The implementation seems to be vulnerable to Coppersmith's attack, where the attacker will solve the above equation for the part of the padded message M that corresponds to the message hash $h(m)$. Since the hash is of size 256 bits, which is less than the limit 341, the attack is likely to succeed. The attack steps are listed below:

Signature Forgery Attack Steps Using Coppersmith's Method

1. Choose an arbitrary m the attacker wants to forge a signature for.
2. Compute the hash $h(m)$.
3. Compute the padded message M according to the PKCS#1 v.1.5 padding scheme: $M = (\texttt{0x00}||\texttt{0x01}||\texttt{0xFF}\ldots\texttt{0xFF}||\texttt{0x00}||\text{ASN.1 structure}||h(m))$
4. Break the solution S into a known part Δ, composed of the padding prefix up to (and excluding) the hash $h(m)$ and an unknown part x, of size $|x| = |h(m)|$, that we want to solve for: $S = (\texttt{0x00}||\texttt{0x01}||\texttt{0xFF}\ldots\texttt{0xFF}||\texttt{0x00}||\text{ASN.1 structure}||x) = \Delta||x = 2^{|x|}\Delta + x$
5. Apply Coppersmith's method to find a root of the following polynomial: $P(x) = (2^{|x|}\Delta + x)^e - M \equiv 0 \mod n$ This is equivalent to solving for x the equation $(2^{|x|}\Delta + x)^e = M \bmod n$, where $e = 3$.
6. If a solution x is found, then $S = \Delta||x$ is a valid forged signature produced without knowledge of the private key d that would be validated correctly since $S^e = (2^{|x|}\Delta + x)^e = M \bmod n$. Stop.
7. If a solution is not found, repeat from step (1.) with a new message m.

An immediate and easy fix would be to enforce the use of large public exponents, such as 65537. This would make the attack less feasible, but still theoretically possible. Therefore, as a long-term solution, consider switching to the `EMSA-PSS` padding scheme. The latter effectively mitigates the attack due to its randomized padding which makes the structure of the padded message M unpredictable. In particular, it is not possible to break M into a known and unknown part.

Update: *Resolved in pull request #5234 at commit c9243c4. The OpenZeppelin Contracts team prefers to keep the implementation flexible by verifying any exponent with a valid signature. A warning was added to the documentation that exponent 65537 is recommended by the NIST and that lower exponents raise security concerns.*

L-09 Incorrect Addition With Point at Infinity - Phase 2

In the `P256` library, the `_jAdd` function adds two points in Jacobian coordinates. This addition operation is used in the `_preComputeJacobianPoints` and `_jMultShamir` functions to perform the computation of $G \cdot u_1 + P \cdot u_2$, which is the main step for signature verification and public key recovery. Note that G is the curve's generator and P the public key. However, this `_jAdd` function does not properly handle the addition of a point with the point at infinity $\mathcal{O}$ (the identity element). For any point Q, the addition of $Q + \mathcal{O}$ should be Q, whereas `_jAdd` returns the incorrect point $(0, 0, 0)$. This prevents a valid signature from being correctly verified.

For instance, if a message was signed with the private key $N - 1$, where N is the order of the group, then P is equal to $-G$. Thus, the pre-computed point `points[0x05]` is $P + G = -G + G = \mathcal{O}$. When this point is looked up during `_jMultShamir` to be added to the rolling coordinates x, y, z, the loop cycle should basically do a no-op, but instead resets x, y, z to $(0, 0, 0)$. The following table contains all private keys that lead to a pre-computed point at infinity.

Private Key	Lookup Point at Infinity
$N - 1$	$P + G, 2P + 2G, 3P + 3G$
$N - 2$	$P + 2G$
$N - 3$	$P + 3G$
$(N - 1)/2$	$2P + G$
$(N - 3)/2$	$2P + 3G$
$(N - 1)/3$	$3P + G$

Two approaches can be taken to address this issue:

1. Perform a no-op by skipping the `_jAdd` operation in `_jMultShamir` when the pre-computed point is the point at infinity.
2. In the `_jAdd` function, check if one of the points is the point at infinity and return the other.

It is important to note that approach (1.) would leave the `_jAdd` function with this particular bug, but would resolve the issue in the context of this implementation. This is the case because no addition with the point at infinity would be performed since the addition operation is skipped in `_jMultShamir`. Within `_preComputeJacobianPoints` only $2P$ and $3P$ are used as points for further calculations ($2P+G$, $3P+G$, etc.), however, since P is an elliptic curve point part of a group of order N where $N \equiv 1 \mod 2$ and $N \equiv 1 \mod 3$, it means that neither $2P$ nor $3P$ can ever be equal to the point at infinity. Hence, the pre-computed points are safe against the bug described above. Approach (2.) would fix the problem at its root but is therefore more gas intense.

Carefully consider the two approaches mentioned above and adopt one. While doing so, thoroughly document any accepted risks or incorrect behaviors. In addition, consider writing a thorough test suite that validates a correct signature verification for previously affected private keys over random messages.

Update: *Resolved in pull request #5218 at commit 427d074. The OpenZeppelin Contracts team chose to implement the first approach.*

Notes & Additional Information

N-01 Lack of Indexed Event Parameters - Phase 1

To improve the ability of off-chain services to search and filter for specific addresses creating proposals, indexing the `proposer` address in the `ProposalCreated` event of `IGovernor.sol` could be useful. Similarly, indexing `proposalId` in the `ProposalCreated`, `ProposalQueued`, `ProposalExecuted`, `ProposalCanceled`, `VoteCast`, and `VoteCastWithParams` events could be beneficial for filtering specific proposals. However, discussion on `issue 3826` of `openzeppelin-contracts` repository revealed that making changes to any of the events in the `IGovernor.sol` interface could be a breaking change for the integrators decoding these events.

Consider documenting this reasoning in `IGovernor.sol` so that integrators are aware of this limitation.

Update: *Resolved in pull request #5175. Proper documentation has been added to the `IGovernor.sol` interface.*

N-02 Privileged User with Zero Execution Delay Can Re-Execute Operations - Phase 1

In the `AccessManager` contract, a user with zero execution delay can re-execute a scheduled and executed operation, a cancelled operation, or an expired operation. Assigning a role with no execution delay comes with a trust assumption that the user is trusted and will not make any malicious or undesirable changes to the protocol, allowing them to execute any operation regardless of the schedule. However, this is in contrast with the comment above the `execute` function in the `AccessManager` contract which states that `_consumeScheduledOp` guarantees that a scheduled operation is only executed once. If a role has zero execution delay, the role takes precedence over the schedule, allowing the role to execute a scheduled operation more than once.

Consider documenting the precedence of the role over the schedule to correctly reflect this scenario.

Update: *Acknowledged, not resolved. The OpenZeppelin Contracts team stated:*

> *The current behavior of the `AccessManager`'s `execute` function is expected as it allows any account with a privileged role to execute a function regardless of any previously scheduled operation. We acknowledge this behavior but do not think documentation is required since it does not change how developers should interact with this contract.*

N-03 Padding Ignored in `Base64URL` Encoding - Phase 1

The `encodeURL` function of the `Base64` contract purposefully ignores any padding. This implementation is based on RFC-4648 which allows the padding to be skipped for URL/URI encoding since the pad character "=" is typically percent-encoded. As highlighted in this research, depending on the behavior of the decoder, the optionality of padding can introduce malleable outputs when decoding this string.

Consider documenting explicitly that the padding is ignored while encoding the `Base64URL` so that integrating projects are aware and able to modify the decoding functionality wherever necessary.

Update: *Resolved in pull request #5176. The OpenZeppelin Contracts team stated:*

> *The team agrees that it is an issue that mostly depends on the decoder implementation. For this reason, we would like to document it on the decoding side once a Base64URL decoder is implemented in OpenZeppelin Contracts. We are adding a small note to clarify its behavior but have decided not to document it extensively.*

N-04 Inconsistent Annotation for Documentation - Phase 1

Throughout the codebase, inconsistent uses of annotations for referencing documentation from the base contracts were identified. For instance, within `Governor.sol`, in line 99, the `@dev See {IGovernor-name}.` annotation is used, whereas in line 839, the docstrings are inherited via the `@inheritdoc IGovernor` annotation.

The use of `@inheritdoc` annotation is also inconsistent between certain contracts. For instance, in `AccessManager.sol`, the annotation is a single-line comment, whereas in `Governor.sol`, the annotation is in a multi-line comment `/** ... */` format.

To improve code readability, consider using a consistent standard for inheriting documentation throughout the codebase.

Update: *Acknowledged, will resolve. The OpenZeppelin Contracts team stated:*

> *This inconsistency has been discussed in issue 3502. Particularly, this comment details the limitations of using `@inheritdoc` for extending documentation. For these cases, we leverage our documentation engine and use the `@dev See {...}` syntax to point users in the right direction while also writing additional documentation. We acknowledge the inconsistency between single-line comments and `/** ... */` comments and will consider making them consistent in the future.*

N-05 Inconsistent Use of Named Returns - Phase 1

Throughout the codebase, multiple instances of functions having inconsistent usage of named returns were identified:

- In the `AccessManager` contract, multiple functions such as `canCall`, `getAccess`, and `hasRole`, utilize named return variables, whereas others like `expiration` do not.

- In the `ERC2771Forwarder` contract, the `_validate` and `_execute` functions use named return variables, whereas the `verify`, `_recoverForwardRequestSigner`, and `_isTrustedByTarget` functions do not.

To improve code readability, consider using the same return style across all of a contract's functions.

***Update:** Resolved in pull request #5178 and in pull request #5177. The OpenZeppelin Contracts team stated:*

> *Previously, the team agreed to name the returned values when there is more than one. We are documenting this clearly in our guidelines and adding missing names where needed according to this policy.*

N-06 Redundant Function Call in `Checkpoints._insert` - Phase 1

In the `_insert` function of the `Checkpoints` library, the `_unsafeAccess` function is used to retrieve a storage pointer to the struct. However, this retrieval has already been performed at an earlier point in the code.

Consider directly using the previously retrieved storage pointer instead of calling `_unsafeAccess` again.

***Update:** Resolved in pull request #5169 at commit 951b97e.*

N-07 Incorrect Panic Error in `modExp` Functions - Phase 1

The `modExp` function in the `Math` library will revert with an invalid panic error if the `staticcall` to the precompile `modexp` reverts due to running out of gas. The call to the precompile may fail if the operation costs more gas than was provided, resulting in `success` being `false`. This causes a revert with the `DIVISION_BY_ZERO` panic error which misleads library users.

Consider reverting with distinct errors for the case where `m == 0` and when the `staticcall` fails due to an out-of-gas error. The above also applies to the analogous modExp function for fixed-length arguments, though this is less likely to occur.

Update: *Acknowledged, not resolved. The OpenZeppelin Contracts team stated:*

> *Detecting an out-of-gas error is already a difficult task. Although we acknowledge that there is a mismatched panic if the function runs out of gas, there is no other side effect. Future community efforts involve EOF, which removes gas observability. For this reason, we would prefer not to make changes that limit our compatibility with it.*

N-08 Poor Documentation in `SafeERC20` - Phase 1

In `_callOptionalReturn` and `_callOptionalReturnBool` functions of the `SafeERC20` library, there is a check that if some data is returned then the first word must be true (1). However, there is no check that the length of the returned data is exactly one word. This can lead to a scenario in which the target contract does not implement a token interface, but at the same time, has a fallback function which returns some data with 1 in the first returned word. In this case, the library will process the output and not revert.

Consider adding a warning to the code so that integrators are aware of this scenario.

Update: *Acknowledged, not resolved. The OpenZeppelin Contracts team stated:*

> *The referenced functions are currently `private` so they are not available for developers to use. We consider the current documentation to be satisfactory since it already describes that non-reverting calls are assumed to be successful in the corresponding entry points (i.e., `safeTransfer`, `safeTransferFrom`, and `forceApprove`). We will consider documenting this if those functions ever become `internal`.*

N-09 Non-Standardized Declaration of `memory-safe` Assembly - Phase 1

There are two types of memory-safe assembly declarations:

- `assembly ("memory-safe")`
- `/// @solidity memory-safe-assembly`

Both types are used in the codebase, leading to inconsistencies. For example, the `SafeERC20` contract uses the former, whereas the `ERC2771Forwarder` contract uses the latter. Moreover, according to Solidity documentation, the latter is likely to be deprecated.

Consider standardizing the use of the `memory-safe` declaration.

Update: *Resolved in pull request #5172 at commit 04e0df3.*

N-10 Unused Import - Phase 1

Having unused imports negatively affect code quality.

In `IAccessManager.sol`, the `IAccessManaged` interface is imported but never used.

Consider removing the unused import statement to improve code clarity and readability.

Update: *Resolved in pull request #5170 at commit 05f7a22.*

N-11 Inconsistent Order Within Contracts - Phase 1

Throughout the codebase, multiple instances of contracts deviating from the Solidity Style Guide due to having inconsistent ordering of functions were identified. The following is a non-exhaustive list of such instances:

- `AccessManager` has mixed orders of `public` and `internal` functions.
- `Checkpoints` has mixed orders of `internal` and `private` functions.

To improve the project's overall legibility, consider standardizing ordering throughout the codebase as recommended by the Solidity Style Guide (Order of Functions).

Update: *Acknowledged, not resolved. The OpenZeppelin Contracts team stated:*

> *A negative aspect of reordering functions is that it may change the compiler output, leading to unexpected results (e.g., suddenly hitting stack-too-deep errors). Also, changing the order is difficult to review and audit as it produces a big diff that may not be substantial. For these reasons, we decided not to make changes to function ordering.*

N-12 Typographical Errors - Phase 1

Throughout the codebase, multiple instances of typographical errors were identified:

- In line 690 of `AccessManager.sol`, the variable name `isTragetClosed` should be `isTargetClosed`.
- In line 235 of `Math.sol`, `expect` should be `except`.

- In line 61 of `SignedMath.sol`, `bytes(0)` should be `bytes32(0)`.
- In line 92 of `StorageSlot.sol`, `an BooleanSlot` should be `a BooleanSlot`.
- In line 102 of `StorageSlot.sol`, `an Bytes32Slot` should be `a Bytes32Slot`.
- In line 112 of `StorageSlot.sol`, `an Uint256Slot` should be `a Uint256Slot`.
- In line 132 and line 142 of `StorageSlot.sol`, `an StringSlot` should be `a StringSlot`.
- In line 152 and line 162 of `StorageSlot.sol`, `an BytesSlot` should be `a BytesSlot`.

To improve code readability, consider fixing the above along with any other instances of typographical errors in the codebase.

Update: *Resolved in pull request #5171 at commit 3e44eed.*

N-13 Inconsistent Declaration of `memory-safe` Assembly - Phase 2

There are two types of memory-safe assembly declarations:

- `assembly ("memory-safe")`
- `/// @solidity memory-safe-assembly`

Both types are being used in the codebase, leading to inconsistencies. For example, the `Packing` contract uses the former, whereas the `Panic` contract uses the latter. Moreover, according to Solidity documentation, the latter is likely to be deprecated.

Consider standardizing the use of the memory-safe declaration.

Update: *Resolved in pull request #5172. The OpenZeppelin Contracts team stated:*

> *Issue N-09 was resolved by migrating the memory-safe declarations to the newer `assembly ("memory-safe")` syntax.*

N-14 Inconsistent Order Within Contracts - Phase 2

Throughout the codebase, multiple instances of inconsistent ordering of functions were identified:

- In the `GovernorCountingFractional` contract, `view` functions should come before `pure` functions and `internal` functions should come before `internal view` functions.
- In the `Heap` library, `internal` functions should come before `internal view` functions.
- In the `P256` library, `internal` functions should come before `private` functions.
- In the `ReentrancyGuardTransient` contract, `internal` functions should come before `private` functions.
- In the `ERC20TemporaryApproval` contract, `public` functions should come before `internal` functions.

To improve the project's overall legibility, consider standardizing ordering throughout the codebase as recommended by the Solidity Style Guide (Order of Functions).

Update: *Acknowledged, not resolved. The OpenZeppelin Contracts team stated:*

> *We generally adhere to the Solidity style guide on function ordering. However, in the presented cases, changing the order of the functions makes it difficult to review and audit parts of the contracts. For this reason, we decided not to change the function ordering in these cases.*

N-15 Missing Named Parameters in Mappings - Phase 2

Since Solidity 0.8.18, developers can utilize named parameters in mappings. This means mappings can take the form of `mapping(KeyType keyName => ValueType valueName)`. This updated syntax provides a more transparent representation of a mapping's purpose.

Within `GovernorCountingFractional.sol`, the `_proposalVotes` state variable can benefit from a named parameter.

Consider adding named parameters to mappings in order to improve the readability and maintainability of the codebase.

Update: *Resolved in [pull request #5204](). The OpenZeppelin Contracts team stated:*

> *We agree that named parameters in mappings improve readability when the mapping `key` specifies a name. In the case of the mapping's `value`, its name is already that of the mapping itself (e.g., `_proposalVotes`). For this reason, we are updating `GovernorCountingFractional` to name the `_proposalVotes` parameter, but we decided not to name its value as we have consistently not done so across the library.*

N-16 Typographical Errors - Phase 2

Throughout the codebase, multiple instances of typographical errors were identified:

- In line 21 of `IERC1363Receiver.sol`, "The address which are tokens transferred from" should be "The address which the tokens are transferred from".
- In line 19 of `ERC1155Utils.sol`, "if the target address is doesn't contain code" should be "if the target address doesn't contain code".
- In line 47 and line 314 of `Heap.sol`, "Binary heap that support values" should be "Binary heap that supports values".
- In line 240 and line 507 of `Heap.sol`, "leafs" should be "leaves".
- In line 12 of `RSA.sol`, "semanticaly" should be "semantically".
- In line 30 of `RSA.sol`, "according the verification" should be "according to the verification".
- In line 140 of `RSA.sol`, "safetiness" should be "safeness".
- In line 95 of `CircularBuffer.sol`, "elements kepts in" should be "elements kept in".

Update: *Resolved in pull request #5194. The OpenZeppelin Contracts team stated:*

> *The indicated typographical errors have been addressed as per the recommendations.*

N-17 Potential Licensing Conflict - Phase 2

The following two cryptography libraries might face licensing conflicts:

- The `RSA` library is said to be inspired by the `adria0/SolRsaVerify` repository, which is licensed under GPL version 3.
- The `P256` library is said to be based on the `itsobvioustech/aa-passkeys-wallet` repository, also licensed under GPL version 3.

As the OpenZeppelin contracts library is released under the MIT license, consider clarifying that inspired and copied code under GPL v3 can be released under MIT.

Update: *Resolved in pull request #5205. The OpenZeppelin Contracts team stated:*

> *The team recognizes the potential licensing conflicts and we are clarifying that the code that inspired our implementation is under a GPL v3 license.*

N-18 Incorrect and Misleading Documentation - Phase 2

Throughout the codebase, multiple instances of incorrect or misleading documentation were identified:

- The documentation of the `VestingWalletCliff` `constructor` does not match the implementation. There is just the `cliffSeconds` parameter to be described. Note that the inherited `VestingWallet` contract also sets up the beneficiary as the owner using the `Ownable` contract instead of using the `Ownable2Step` contract as indirectly described.
- The title of the `IERC1363Spender` is incorrect as it is missing the "I" prefix.
- In `P256.sol`, the comment for `isValidPublicKey` states $x \leq P, y \leq P$, whereas it should be $x < P, y < P$. Note that x, y are coordinates of an EC point and P is the modulus of the base field they are elements of. So, all values of the coordinates are strictly less than the modulus.
- In `P256.sol`, the comment for `_jMultShamir` says $Pu_1 + Qu_2$, whereas it should be $Gu_1 + Qu_2$.
- The `_preComputeJacobianPoints` function has some comments on the end-result of each table entry. For consistency, the `0x0c` array entry should be commented as `(3g)` instead of `(g+2g)`. Also note that one index is `0x0C` with a capital "C" instead of the otherwise lowercase hex characters.
- In RSA signature schemes, the `DigestInfo` allows having some optional parameters. However, these are not checked in the `RSA` implementation. In principle, this may create a malleability issue. Although, the way the `buffer` is parsed suggests that the implementation does not support optional parameters. Consider clarifying this in the documentation.
- In `RSA.sol`, the verification has the sha-256 OID hard-coded in it, which is the only hash function supported at the moment. Since there is a wrapper (`pkcs1Sha256`) for SHA256, it suggests the possibility of adding wrappers for other hash functions (e.g., Keccak) as well. If another hash function is used with the verification procedure, verification will fail due to the hard-coded OID. Consider clarifying in the documentation that the implementation currently only supports SHA256.

- In the `pkcs1` function of `RSA`, the length of the RSA modulus n is compared to the constant `0x40`. However, it is not clear how this constant is computed. Consider clarifying this in the documentation.
- The index description of the `CircularBuffer` library describing "The last item is at [...] and the last item is at [...]" is unclear.

Consider applying the above suggestions for a clearer documentation that helps reason about the codebase.

Update: *Resolved in pull request #5206 at commit 0a0d44d and in pull request #5229 at commit 9c986c5. The OpenZeppelin Contracts team stated:*

> *The recommendations were implemented with a few differences. In the case of `P256.sol`, it was renamed to `G·u1 + P·u2` instead of `G·u1 + Q·u2`, which is consistent with the arguments of `_preComputeJacobianPoints`. For `RSA.sol`, the `pkcs1` function was renamed to `pkcs1Sha256` to be consistent with the hard-coded OID. Also, the RSA modulus is now enforced to be at least `0x100` following the recommendation from N-22.*

N-19 Redundant Code - Phase 2

Within the `recovery` function of the `P256` library, the `v % 2` operation is unnecessary given that the value of `v` can only be 0 or 1.

Consider removing any redundant code to improve the readability and maintainability of the codebase.

Update: *Resolved in pull request #5200.*

N-20 Inconsistent Integer Base Within a Contract - Phase 2

In `RSA.sol`, integer constants are represented using both decimal (in line 98 and line 104) and hexadecimal (in line 49, line 56, line 63, line 98, line 104, and line 134).

To avoid confusion and improve the readability of the codebase, consider using a consistent notation.

Update: *Resolved in pull request #5206.*

N-21 Inconsistent `_jAdd` Function Interface - Phase 2

In the `_jAdd` function of the `P256` library, the first point is passed as a `JPoint` struct, while the second point is passed with the explicit coordinates. In the assembly block, this leads to fetching the x, y, and z coordinates from the first point using the `mload` and `add` opcodes, thereby making the code additionally complex.

Consider changing the function interface by passing both points' coordinates explicitly. Alternatively, consider documenting why this interface is required as is.

Update: *Acknowledged, not resolved. The OpenZeppelin Contracts team stated:*

> *Using `JPoint` for both points would be significantly more gas-expensive in the context of how `_jAdd` is used within other functions, particularly `_jMultShamir`. The current design minimizes unnecessary memory operations. The `_jAdd` function is primarily called from contexts where one point is already in `JPoint` format (often a running calculation result), while the other is a new point being added, for which we have direct coordinate access (such as in `_jAddPoint`).*
>
> *Note that `_jAdd` using mixed input (point in memory + coordinates on the stack) is necessary to circumvent stack-too-deep errors. The mloads for point 1 are done at the very last moment. Doing the mload only once and caching the value on the stack would possibly save gas, but it also blocks compilation without via-ir.*

N-22 Arbitrary RSA Modulus Size - Phase 2

The comment for `pkcs1` correctly states that using an RSA modulus of `1024` bits is unsafe and encourages the use of at least `2048` bits.

Consider enforcing this requirement in the code so that the signatures produced under moduli of size less than `2048` are not supported. This is in accordance with the minimum key sizes recommended by NIST throughout the year 2030.

Update: *Resolved in pull request #5206. The OpenZeppelin Contracts team stated:*

> *We acknowledge the risks of using a modulus of less than `2048` bits according to the documentation so we are enforcing it.*

N-23 Custom Functions Might Modify Memory - Phase 2

The custom `fnHash` function of the `MerkleTree` library and the `comp` comparator function of the `Heap` library are arbitrary functions passed as input parameters by integrators. There are no restrictions on these functions apart from the list of input and output parameters. As such, integrators might maliciously or accidentally code these functions in a way that modifies the memory state whenever they are executed. Depending on the logic of these functions, modifications done to memory can result in unexpected behavior.

While balancing the trade-offs between providing flexible library code and the risk of side effects like memory manipulation, consider adding the aforementioned edge case as a warning in the documentation.

***Update:** Resolved in pull request #5213 and in pull request #5190.*

N-24 Lack of Input Validation - Phase 2

The `setup` function of `CircularBuffer.sol` does not verify if the `size` of the buffer is zero. In addition, the `push` function of `CircularBuffer.sol` does not verify if the buffer `self` was set up. This could potentially lead to the `modulus` being zero and revert due to modulo by zero.

Consider checking the inputs explicitly to reduce the attack surface of the codebase.

***Update:** Resolved in pull request #5214. A check was added to the `setup` function to ensure that the buffer size is not zero. In the `push` function, the panic of modulus by zero has been kept as it is.*

N-25 Unintialized Variable - Phase 2

Initializing some variables and leaving out others can impact code readability. In the `_countVote` function of the `GovernorCountingFractional` contract, the `usedWeight` variable has not been initialized whereas others have been.

To improve code clarity and readability, consider initializing the `usedWeight` variable as well.

***Update:** Resolved in pull request #5206.*

N-26 Applicability of Padding Oracle Attacks to `RSA.sol` - Phase 2

The `EMSA-PKCS1-v1_5` padding scheme has been shown to be susceptible to implementation errors that allow signature forgeries [1, 2] using variants of the padding oracle attack by Bleichenbacher (1998), originally proposed for RSA encryption. In a padding oracle attack on signatures, the attacker submits carefully chosen malformed signatures, modifying parts of the padding. From the responses it gets from the signing algorithm, the attacker is able to produce a forgery with high probability.

As a mitigation, the RFC8017 standard proposes the EMSA-PSS padding scheme. This scheme is probabilistic, meaning that the same message can have different paddings each time it is signed. The added randomness effectively mitigates padding oracle attacks. In addition to having oracle access to the verification algorithm, the Bleichenbacher attack on signatures leverages the following weaknesses:

1. Faulty implementations that only check the first bytes of the decrypted signature (`0x00`, `0x02`, and the `0xFF` padding) and omit to check the rest of the formatting. In particular, they fail to check if the hash data matches the hash of the supplied message.
2. Use of small values for the public exponent $e = 3$. Although the attack is theoretically possible for large values of e (e.g., 65537), it is much less efficient in that case.
3. Malleability of the RSA scheme, i.e., small changes to the ciphertext or signature result in predictable changes in the plaintext.

Point (1.) is not applicable to the `RSA.sol` implementation due to the thorough check being performed on the full decrypted signature. On the other hand, points (2.) and (3.) are applicable. As a result of the mitigation of point (1.), the padding oracle attack in its original form is deemed as inapplicable to the specific implementation in `RSA.sol`.

The above being said, the `EMSA-PSS` padding scheme enforces a full check of the format of the decrypted signature (point 1.). It also binds the hash, the randomness, and the padding together, which makes any manipulation of the signature easily detectable. Moreover, it fixes the inherent malleability property of RSA (point 3.), making the scheme non-malleable. Therefore, in the interest of long-term security, consider switching to the `EMSA-PSS` padding scheme.

***Update:** Acknowledged, not resolved. The OpenZeppelin Contracts team stated:*

> *Indeed, EMSA-PSS looks like a better, more modern scheme than EMSA-PKCS1-V1_5 which we currently support. We will consider adding support for EMSA-PSS in future releases to improve the coverage of our library. That being said, it is going to take time*

> *to implement, test, and audit. In the meantime, EMSA-PKCS1-V1_5 is what we will support.*

N-27 Inconsistent Documentation - Phase 2

The docstring above the `MerkleTree` library states that the library has been *available since v5.1*. However, this statement is missing from other contracts that are introduced in this version release.

For consistency, consider either adding this statement to all the contracts or removing it from the `MerkleTree` library.

Update: *Acknowledged, not resolved. The OpenZeppelin Contracts team stated:*

> *These release notes have been added to previous versions since we found them to be relevant in some cases. While the addition of the release notes has not been automated, we decided to leave this note and make it more consistent in future documentation updates.*

N-28 Naming Suggestions - Phase 2

Throughout the codebase, multiple instances of inconsistent or unclear naming were identified:

- The `MerkleTree` library refers to the height of the binary tree both as "depth" [1, 2, 3] and "levels" [4, 5, 6, 7]. Consider sticking to one word for consistency.
- In the `Heap` library documentation, node `i` is *referred to as father* for the nodes at index `2*i+1` and `2*i+2`. Consider referring to `i` as the *parent* node for a more neutral wording.
- In the `pop` function of `Heap`, the first node of the data array is saved in the `rootNode` variable, while the actual root node is saved as `rootData`. Consider renaming `rootNode` `firstDataNode`, `rootData` `rootNode`, and `lastNode` to `lastDataNode` for a more descriptive name.

Consider implementing the above-mentioned naming suggestions to improve the readability and maintainability of the codebase.

Update: *Resolved in pull request #5215. The third point was resolved with L-07.*

Client Reported

CR-01 Incorrect Internal Call - Phase 1

During the audit, the OpenZeppelin Contracts team detected that the `verifyCallData` and `multiProoVerifyCalldata` functions of the `MerkleProof` library incorrectly call `processProof` and `processMultiProof` internally instead of calling the corresponding calldata variants `processProofCalldata` and `processMultiProofCalldata`.

***Update:** Resolved in pull request #5140 at commit 24a641d.*

CR-02 Incorrect `_jAdd` Result When a Point Is Added to Itself - Phase 2

This bug was brought to our attention by Zellic. When the two points input to the `_jAdd` function in `P256.sol` are equal, the function returns the incorrect result $(0, 0, 0)$.

The bug can be triggered along two paths:

1. `verifySolidity` calls `_preComputeJacobianPoints` calls `_jAdd`
2. `verifySolidity` calls `_jMultShamir` calls `_jAdd`

The analysis along path 1 identifies 7 distinct valid weak private keys d that would trigger the bug: $d \in \{1, 2, 3, 2^{-1}, 3^{-1}, 2\ 3^{-1}, 3\ 2^{-1}\}$. Assuming that d is chosen uniformly at random among $n - 1$ values, where n is a 256 bit prime, the probability to choose any one of the weak keys is negligible: $\frac{7}{2^{256}-1} \approx 0$.

Along path 2, at most 2048 valid weak private keys d are estimated that would trigger the bug. While significantly larger than the path 1 case (7), the size of this set is still negligible w.r.t. the size of the whole space 2^{256}. Specifically, the probability of randomly drawing one of the weak keys is at most: $\frac{2048}{2^{256}-1} \approx 0$.

This analysis is in accordance with Zellic's security reduction argument which states:

> If an attacker $\mathcal{A}$ chooses a (weak) public key and "manages" to compute $\sigma = (r, s, h)$ that passes verification, then he'll also be able to efficiently compute the corresponding (weak) private key d from the public key, which will invalidate the hardness of ECDLP assumption.

In terms of attack scenarios in which the bug can be exploited, we analyzed the following scenarios (including the two scenarios reported by Zellic):

1. Zellic scenario 1: $\mathcal{A}$ holds a private key and valid signatures. However, these get erroneously rejected by `verifySolidity`. In detail, $\mathcal{A}$ chooses a weak public/private key pair and produces a valid signature $\sigma = (r, s, h)$. The signature $\sigma$ will be rejected due to the bug.
2. Zellic scenario 2: $\mathcal{A}$ holds a private key with invalid signatures. However, these get erroneously accepted by `verifySolidity`.
3. $\mathcal{A}$ holds a private key with a valid signature. However, it gets erroneously accepted by `verifySolidity` with a public key that does not match the private key by triggering the bug.

In summary, the reported bug looks highly unlikely to be triggered by chance. It also seems that the possibilities to exploit it maliciously are very limited and do not have critical consequences. That being said, it should be noted that if there are other attack scenarios apart from the ones listed above, they would require further investigation. In addition, the above analysis assumes that the bug can only be triggered from the `verifySolidity` function as an entry point. Last but not least, the bug clearly identifies an error in the computation, regardless of any security implications. Therefore, it is recommended to change the `_jAdd` implementation to cover the case where a point is added to itself, as outlined in the Zellic write-up.

Update: *Resolved in pull request #5218 at commit d3b67ce.*

Recommendations

Protection Against Postquantum Adversaries in RSA Signatures

In view of the fact that blockchain data is public and persists forever, attention can be drawn to the possibility of adding perfect forward secrecy (PFS) to RSA signatures. This would be done to prevent the recovery of the private key of a signature from the public key at a point in the future when computational capabilities are more advanced (e.g., in a post-quantum setting).

PFS is typically associated with key exchange protocols (e.g., DH/ECDH) where session data is encrypted with a temporary (ephemeral) session key which is discarded at the end of the session. In case the master private key that was used to derive the session key (e.g., using a hash function) is compromised, the encrypted data that was transmitted in the past cannot be decrypted in the future.

Adding PFS to signatures is uncommon and, to the best of our knowledge, no signature schemes exist with this property. Indeed, such functionality would invalidate one of the core properties of digital signatures, namely non-repudiation, which ensures that a signer is not able to deny signing a piece of data after the fact. On the other hand, the relatively recent emergence of blockchains presents new use cases that may, to some extent, justify the suggested modification.

Specifically, the fact that blockchain data is public and persists forever may allow post-quantum adversaries to recover the private key of a signature from the public key using quantum computers. With PFS, the negative consequences from the latter will be prevented by having an analogous notion of temporary private keys for signing that would be valid for a long, but fixed, period of time (say, 10-20 years).

We suspect that adding PFS to digital signatures may require some non-trivial changes to the signing and verification process. Yet, the changes would probably be more on the signer's side, while the verifier would just need to discard signatures produced under invalid (i.e., expired) keys.

Conclusion

The v5.1 release of OpenZeppelin Contracts introduces several significant enhancements, including the addition of a cliff period to the vesting wallet, support for fractional voting in governance, implementations of the ERC-1363 and ERC-7674 standards, as well as new implementations of data structures and cryptographic libraries. We commend the Solidity Contracts team for addressing user needs and their efforts to improve existing features while introducing new utilities.

Throughout the six-week engagement, three medium-severity vulnerability were identified. Furthermore, several recommendations were provided to adhere to best practices and minimize the attack surface. Special attention was given to documenting edge cases to ensure that integrators are aware of potential risks when interacting with these contracts. These efforts are intended to foster the development of a more resilient codebase keeping in mind the library's significance as a foundational element in the blockchain ecosystem. We are again very grateful for the opportunity to collaborate with the Contracts team on this next milestone.