OpenZeppelin | security

Contracts 5.0 Release

October 3, 2023

Table of Contents

Summary

Type	Library
Timeline	From 2023-06-05 To 2023-09-15
Languages	Solidity

Total Issues	90 (49 resolved, 16 partially resolved)
Critical Severity Issues	0 (0 resolved)
High Severity Issues	3 (1 resolved)
Medium Severity Issues	11 (7 resolved, 2 partially resolved)
Low Severity Issues	17 (8 resolved, 2 partially resolved)
Notes & Additional Information	55 (29 resolved, 12 partially resolved)
Client-Reported Issues	4 (4 resolved)

Scope

Phase 1

We audited the OpenZeppelin/openzeppelin-contracts repository at commit 99a4cfc and the following files were in scope:

```
contracts
├── ERC20
│   ├── ERC20.sol
│   ├── IERC20.sol
│   ├── extensions
│   │   ├── ERC20Burnable.sol
│   │   ├── ERC20Capped.sol
│   │   ├── ERC20FlashMint.sol
│   │   ├── ERC20Pausable.sol
│   │   ├── ERC20Permit.sol
│   │   ├── ERC20Votes.sol
│   │   ├── ERC20Wrapper.sol
│   │   ├── ERC4626.sol
│   │   ├── IERC20Metadata.sol
│   │   └── IERC20Permit.sol
│   └── utils
│       └── SafeERC20.sol
├── ERC1155
│   ├── ERC1155.sol
│   ├── IERC1155.sol
│   ├── IERC1155Receiver.sol
│   ├── extensions
│   │   ├── ERC1155Burnable.sol
│   │   ├── ERC1155Pausable.sol
│   │   ├── ERC1155Supply.sol
│   │   ├── ERC1155URIStorage.sol
│   │   └── IERC1155MetadataURI.sol
│   └── utils
│       ├── ERC1155Holder.sol
│       └── ERC1155Receiver.sol
└── utils
    ├── StorageSlot.sol
    └── structs
        └── Checkpoints.sol
```

The following contracts were audited at the 99a4cfc commit but only the changes between versions 5.0 and 4.9 were in scope:

```
contracts
└── utils
    └── math
```

```
        ├── Safecast.sol
        └── Math.sol
```

Phase 2

We audited the OpenZeppelin/openzeppelin-contracts repository at commit 8fff875 and the following files were in scope:

```
contracts
├── interfaces
│   ├── draft-IERC1822.sol
│   └── IERC1967.sol
├── proxy
│   ├── beacon
│   │   ├── BeaconProxy.sol
│   │   ├── IBeacon.sol
│   │   └── UpgradeableBeacon.sol
│   ├── ERC1967
│   │   ├── ERC1967Proxy.sol
│   │   └── ERC1967Utils.sol
│   ├── Proxy.sol
│   ├── transparent
│   │   ├── ProxyAdmin.sol
│   │   └── TransparentUpgradeableProxy.sol
│   └── utils
│       └── UUPSUpgradeable.sol
├── metatx
│   ├── ERC2771Context.sol
│   └── ERC2771Forwarder.sol
├── finance
│   └── VestingWallet.sol
└── utils
    ├── Address.sol
    ├── Context.sol
    ├── cryptography
    │   └── EIP712.sol
    └── Strings.sol
```

Phase 3

We audited the OpenZeppelin/openzeppelin-contracts repository at commit b027c35 and the following files were in scope:

```
contracts
├── access
│   ├── AccessControl.sol
│   ├── AccessControlDefaultAdminRules.sol
│   ├── AccessControlEnumerable.sol
│   ├── Ownable.sol
│   └── Ownable2Step.sol
├── proxy
```

```
│       └── Clones.sol
├── security
│   └── Pausable.sol
├── token
│   ├── common
│   │   └── ERC2981.sol
│   ├── ERC721
│   │   ├── ERC721.sol
│   │   ├── IERC721.sol
│   │   ├── IERC721Receiver.sol
│   │   ├── extensions
│   │   │   ├── ERC721Burnable.sol
│   │   │   ├── ERC721Consecutive.sol
│   │   │   ├── ERC721Enumerable.sol
│   │   │   ├── ERC721Pausable.sol
│   │   │   ├── ERC721Royalty.sol
│   │   │   ├── ERC721URIStorage.sol
│   │   │   ├── ERC721Votes.sol
│   │   │   ├── ERC721Wrapper.sol
│   │   │   ├── IERC721Enumerable.sol
│   │   │   └── IERC721Metadata.sol
│   │   └── utils
│   │       └── ERC721Holder.sol
└── utils
    └── cryptography
        └── MessageHashUtils.sol
```

The following contracts were audited at the b027c35 commit but only the changes between versions 5.0 and 4.9 were in scope:

```
contracts
└── utils
    ├── Arrays.sol
    ├── Base64.sol
    ├── Create2.sol
    ├── cryptography
    │   └── ECDSA.sol
    ├── introspection
    │   ├── ERC165.sol
    │   ├── ERC165Checker.sol
    │   └── IERC165.sol
    └── structs
        └── BitMaps.sol
```

Phase 4

We audited the OpenZeppelin/openzeppelin-contracts repository at commit b5a3e69 and the following files were in scope:

```
contracts
├── access
│   └── manager
```

```
│       ├── AccessManaged.sol
│       ├── AccessManager.sol
│       ├── IAccessManager.sol
│       ├── IAccessManaged.sol
│       └── IAuthority.sol
└── utils
    └── types
        └── Time.sol
```

The following contracts were audited at the b5a3e69 commit but only the changes between versions 5.0 and 4.9 were in scope:

```
contracts
└── utils
    └── math
    │    └──  SignedMath.sol
    ├── structs
    │   ├── EnumerableMap.sol
    │   └── EnumerableSet.sol
    └── Multicall.sol
```

Phase 5

We audited the OpenZeppelin/openzeppelin-contracts repository at commit adbb8c9 and the following files were in scope:

```
contracts
├── proxy
│      └── utils
│            └── Initializable.sol
├── security
│     └── ReentrancyGuard.sol
├── governance
│   ├── extensions
│   │   ├── GovernorCountingSimple.sol
│   │   ├── GovernorSettings.sol
│   │   ├── GovernorStorage.sol
│   │   ├── GovernorTimelockAccess.sol
│   │   └── GovernorTimelockControl.sol
│   ├── Governor.sol
│   ├── TimelockController.sol
│   └── utils
│       └── Votes.sol
└── utils
    ├── cryptography
    │   └── MerkleProof.sol
    └── structs
        └── DoubleEndedQueue.sol
```

The following contracts were audited at the adbb8c9 commit but only the changes between versions 5.0 and 4.9 were in scope:

```
contracts
└── governance
    └── extensions
        ├── GovernorVotes.sol
        ├── GovernorVotesQuorumFraction.sol
        ├── GoverorPreventLateQuorum.sol
        └── GovernorTimelockCompound.sol
```

Overview

The major version 5.0 of this library brings many breaking and important changes. Before diving into each phase's changeset description, the general applicable changes include:

- A switch from `require` statements to custom errors instead. Custom errors have been introduced with the latest Solidity versions and are a natural evolution of error handling within Solidity. `require` statements had to be managed with custom strings to describe the error, introducing difficulties and error-prone techniques; custom errors on the contrary are more explicit and easy to handle, making the code more readable and robust. The team is also behind the draft of EIP-6093 in an effort to standardize custom errors being used for common token implementations. Finally, custom errors appear to be more convenient in terms of gas costs as one can read from a benchmark test that the team performed.
- Updated the Solidity version to 0.8.20. This sparked a discussion about the compatibility with chains that lack adoption of the `PUSH0` opcode. You can read more about the needs and consequences here.
- Some contracts were removed, like `Counters`, and some others have been added such as `Nonces`.
- The code style went under a refactor that greatly improved consistency across all files. Some examples are contracts that had to be marked as abstract when they were not, contracts that were a good fit to be defined as library instead and modifiers that now are consistently using internal functions instead of having the logic directly within their definition.

There is a more detailed and technical changelog here. Below is a brief breakdown of the focus and the corresponding set of changes for each individual phase.

In Phase 1, we focused mainly on token standards ERC-20 and ERC-1155. The main changes are in the inner mechanics. The `_update` internal function is now in charge of summarizing the contracts' token accounting logic when it comes to transfers, minting/burning and

allowances checks. The `_update` function replaced many internal functions that were previously used to handle the same logic. With this change, the team believes in better maintenance of the code while having a more modular and compact code. Moreover, the legacy transfer hooks are now removed and replaced by overrides of the `_update` function, making custom integrations easier to perform.

In Phase 2, we switched the focus to proxies and their related contracts. One change is that `ERC1967Upgrade` is no longer an abstract contract, and is now a library, renamed to `ERC1967Utils`. Apart from this, there is an important change in design in which the `TransparentUpgradeableProxy` now deploys its own `ProxyAdmin`, assigning it to an `immutable` `_admin` variable so that there will be a new proxy admin for every transparent proxy with no possibility to change it. The same thing occurs with beacons, which are immutable and cannot be changed once set.

Phase 3 was about auditing ERC-721 and some basic access contracts in preparation for a big change in Phase 4. The ERC-721 part was similar to Phase 1, where the most important changes are the use of the `_update` function and the removal of before and after hooks. Notice that even if before and after transfer hooks were removed, the check on ERC-721 received is still supported, given the fact that it serves an explicit and separate security concern. In this phase, we also inspected the `Ownable` contract, where a notable change is that the contract does not default to `_msgSender` as the initial owner anymore, and an `initialOwner` parameter must be specified upon construction.

In Phase 4, we moved to one of the main features of this new major version: the `AccessManager` contract. The `AccessManager` can be seen as a more sophisticated access management solution combining and extending the concepts of `AccessControl` and `TimelockController`. Some of its features are:

- Access management rules are defined per function selector for each managed contract. The rules are defined and handled altogether in the `AccessManager` contract for any set of contracts.
- Each restricted function is linked to a group allowed to access it. A group essentially defines a role. Members are granted access to a group by the group's admin. The group linked to a function can be changed by an authorized entity.
- Each member is configured with two kinds of delays:
    - A grant delay, which is the time period that needs to pass from the time point when the member is granted access until the time point that the member becomes active. All members of a group share the same grant delay.
    - An execution delay, which is a timelock-like delay defining the time period that needs to pass from when the member schedules the call to a restricted function

until the time point from which on the member is allowed to actually call that function.

- Delay mechanisms are defined for the administration tasks within the `AccessManager` contract. All sensitive operations (e.g., setting the grant delay for a group, changing a group's admin, changing the group linked to a restricted function) are subject to a delay mechanism.
- The relay functionality allows deployed `Ownable` contracts to migrate to an `AccessManager` instance by transferring the ownership to `AccessManager` and relaying the restricted functions calls through it (and similarly for deployed `AccessControl` contracts).

While it can be difficult to approach because of the increased complexity, the `AccessManager` provides a more flexible management system residing in a unique central contract, while featuring relays and administrations in an entire system. Additionally, during this phase, the `Time` contract has been audited. This new contract defines a `Delay` type to represent duration (delay) that can be configured to change value at a given time point automatically.

Finally, in Phase 5, we focused on the governance module. The governance module has been modified to integrate a new `GovernorTimelockAccess` extension contract, to enable compatibility with `AccessManaged` contracts. The added value here is that proposal targets that are subject to `AccessManager` delays do not require separate proposals for scheduling and executing the target. In addition, the `GovernorStorage` extension contract has been introduced, which enables storing the proposals' details in storage as well as proposal enumerability.

Update: ,
. / , `AccessManager` ' `groups`
`roles` `relay` `execute`.

. / , #4644
.

Security Considerations and Threat Model

As security researchers, our focus has primarily been on auditing protocols within the blockchain ecosystem, ensuring their robustness and resilience against potential vulnerabilities. However, in recent times, this scope has expanded to include the examination of libraries - foundational code that serves as a basis for developers to extend and build upon. With this new area of investigation, we have come to realize that the traditional severity definitions, which were initially designed for protocols, may not fully cover the vulnerabilities that libraries can introduce.

In libraries, vulnerabilities can take on different forms. First, there are explicit flaws where specific parts of the library, such as contracts or functions, do not perform as intended. These issues can directly impact the security of the library and, by extension, the protocols built upon it.

However, challenges arise with implicit vulnerabilities. While the individual components of the library may work properly in isolation, combining multiple contracts or engaging in certain actions, such as overriding virtual functions, can unexpectedly introduce security risks. This suggests that the vulnerability lies not with the developer's implementation but rather with the design of the library itself. You can read about those at the end of this report.

During this audit, our approach has been to try to simulate use cases and classic user patterns and mistakes when it comes to developing smart contracts. While we approached the codebase from many different angles, we did not lose the focus on assessing the correctness of the code itself. Finally, the OpenZeppelin Contracts library has been a pillar for many projects in the past and many more yet to come, and for this, we also tried to bring recommendations and areas of research and study for the team to bring its adoption and security even further.

About severity classifications, we used this as a reference guide:

- : The issue significantly jeopardizes the security of the protocol built upon the library, leading to a high risk of compromising sensitive information, causing substantial financial losses, or severely damaging the protocol's reputation.

- : The issue poses a substantial risk to the security of the protocol built upon the library, potentially compromising sensitive information, causing temporary disruptions, or resulting in moderate financial losses.
- : The issue presents a moderate risk to the security of the protocol built upon the library, potentially affecting a subset of users, having a moderate financial impact, or having a workaround.
- : The issue represents a relatively minor risk to the security of the protocol built upon the library, typically with limited or infrequent exploitation potential. It may involve non-security related concerns worth noting to enhance the codebase's quality.
- *&* : This category encompasses non-security relevant issues that are worth noting to improve the codebase's overall quality, irrespective of their direct impact on the security of the protocol.

Finally, the methodology we adopted also included pre-audit threat modelling sessions between the security services team and the contracts team to investigate particular angles of attack on the contracts and create priorities about what to focus on when auditing specific contracts. These have been fruitful conversations where the auditors had the opportunity to learn the motivations behind particular code designs and patterns. The contracts team was provided with increased visibility over the typical attack vectors and scenarios that are prepared when assessing the security of the contracts.

High Severity

H-01 Potential Inaccuracies in Voting Unit Accounting When Overriding the `ERC20Votes#_getVotingUnits` Function's Formula

The `_getVotingUnits` function in the `ERC20Votes` contract determines the voting power of an account, typically based on its token balance. This function is designed to be overridden, allowing developers to modify the voting power system, such as implementing quadratic voting.

However, if this function is overridden and the formula is altered, there is a potential issue with the accounting of voting units during token transfers, particularly when the token holder delegates their votes to themselves or another address.

The problem arises because the `_update` function in the `ERC20Votes` contract invokes the `_transferVotingUnits` function from the `Votes` contract, passing the transferred token amount as a parameter instead of the corresponding voting units that these tokens represent for either the `from` address or its delegate.

Let's consider a quadratic voting system as an example. Suppose Alice holds 100 tokens and delegates her voting power to Bob. In this scenario, Bob would possess 10,000 voting power units (100^2 = 10,000).

Now, let's assume Alice transfers her 100 tokens to Charlie, who delegates tokens to himself. During the transfer:

- The voting power transferred from Alice to Charlie would be 100 instead of 10,000. This discrepancy occurs because line 112 uses the raw transferred token amount instead of the underlying voting units. As a result, Bob would have 9,900 voting power units, while Charlie would have 100 voting power units, leading to an inaccurate distribution.
- The `_totalCheckpoints` variable, which tracks the total voting power units over time, would also yield incorrect calculations as it utilizes the raw token amount instead of the underlying voting units.

Furthermore, if Charlie re-delegates his tokens to Bob, Bob's new voting power would consist of the remaining 9,900 voting units plus an additional 10,000 voting units, resulting in a total of 19,900 voting power units. This behavior arises because the `_delegate` function in the `Votes` contract correctly utilizes the `_getVotingUnits` function to calculate the voting power units to be transferred.

Exploiting this vulnerability, an attacker could artificially inflate their voting power and potentially seize control of the governance by creating and voting for proposals.

When calling the `_transferVotingUnits` function in the `_update` function of the `ERC20Votes` contract, consider sending as the third parameter `_getVotingUnits(account)` instead of `amount`, and also changing the `from` and `to` as parameters, send `delegates(from)` and `delegates(to)` respectively.

Update: , .

> ' ,
> `_transferVotingUnits` `_update` ,
> .
> `_toVotingUnits(uint256)`
> . ,
> - , ,
> ./ ,
> .

H-02 Non-Compliance of `ERC2771Context` With ERC Could Lead to Incorrect Address Extraction

The implementation of the `_msgSender` function in `ERC2771Context` does not adhere to the specifications laid out by the EIP. Specifically, it fails to retain the original `msg.sender` when `msg.data` is shorter than 20 bytes.

In cases where `msg.data` is less than 20 bytes, the extracted address becomes `address(0)`. This is due to the EVM reading out-of-bounds calldata as zeros. Consequently, `address(0)` will be returned as the `msg.sender`.

Coincidentally, due to a separate vulnerability, if the `refundReceiver` in the forwarder's `executeBatch` method is set to be the receiver contract, it will result in a call that is also not compliant with the EIP, since it does not append the sender's address to the end of the calldata.

In combination with the non-compliant address extraction, if both contracts are used together, and the receiver contract implements a fallback or a receive method that makes use of the `_msgSender()` method, the contract will interpret the sender as `address(0)`. This can result in unexpected consequences, as this address is often used as a default or burn address, and is presumed to be an impossible sender. For example, tokens explicitly or implicitly owned by `address(0)` or contracts with revoked ownership may be exploited. The high likelihood of this occurring, despite the multiple prerequisites, is demonstrated by the fact that both of the needed vulnerabilities co-exist in this codebase.

Consider adhering to the original specifications and returning the original `msg.sender` when `msg.data` is less than 20 bytes.

Update: *#4481 7 712 , #4484 4435 .*

> • `_msgSender()` `msg.sender` 20 .
>
> • `_msgData()` `msg.data` 20 . `_msgData()` `_msgSender()` .

H-03 Risk of Failed L2 and Sidechain Deployments with Solidity Version 0.8.20

The codebase has been updated to Solidity pragma `^0.8.20`. With this version, Shanghai becomes the default EVM version employed by the Solidity compiler and hardhat. This may pose an issue for projects deploying to L2s and non-Ethereum-mainnet chains, most of which don't support Shanghai EVM. Specifically, the compiler will generate bytecode that includes `push0`, which will cause deployment failures on these chains.

Since projects will have to configure their project to use a non-default proactively, a significant number of projects may overlook this new requirement when using the new library version. Crucially, this problem is likely to go unnoticed until the deployment stage. During deployment, most transactions are likely to fail due to the invalid opcode. Besides failed deployments, there's a risk of projects transferring funds or ownership to an address presumed to be deployed, but which contains empty code due to the failed deployment. This is because

the failed deployment transaction will still carry a non-empty `contractAddress` field, and that address will still accept function calls and native token transfers.

While the impact of a failed deployment would be limited in most cases, it will be a disruptive event for a project, requiring investigation and causing deployment delays. Additionally, the impact on the contracts library's reputation could be significant if multiple projects encounter this issue. Given its high likelihood and medium impact, this problem's severity appears high.

Consider maintaining the use of `^0.8.19` until the community is well aware of the risks and tooling has improved to mitigate these risks for most projects.

Update: , .

> *0.8.20* *0.8.20.*
>
> *evmVersion*
>
> *,* *.*
>
> *.* *0.8.20.*

Medium Severity

M-01 Potential Reentrancy in `ERC1155._update` Function

When the `_update` function of the `ERC1155` contract is triggered either by a transfer or a mint/burn, the last action performed in the execution is acceptances checks, where the execution calls the `to` recipient doing an external call in the case it's not an EOA and it's effectively a contract.

This means that the recipient contract must implement `IERC1155Receiver` and answer to the external call properly to let execution flow without errors. If the recipient happens to be a contract that either fails in returning the right data or just fails in returning any data, then the `_update` execution will revert.

This is dictated by the EIP definition. However, when dealing with a smart contract library that makes extensibility the most important feature, one should take care of this external call and avoid falling into the error of failing at the checks-effects-interactions pattern.

Concretely if a contract extends from `ERC1155` and overrides the `_update` function in a way that it performs state updates after the `super._update` call, the external call might be used

maliciously to reenter the contract before those state updates are performed, violating the mentioned security pattern.

Consider informing the users in the docstrings of the limitation. One mitigation might be isolating the acceptances checks and so the external call, separately from the `_update` functions so that implementers that override `_update` have the flexibility to perform those external calls at the very end of their execution.

Update: *#4398.*

> `_update`
>
> 1) `_update` .
>
> 2) `_updateWithAcceptanceCheck` `_update` `receive` . `_update` .

M-02 Unhandled Silent Failures

There are some cases in the code base where failures are not handled early nor routed to some explicit error message. This means that execution can unexpectedly stop, finally reverting.

- In the `ERC1155Supply._update` function, the check that `ids` and `amounts` have the same length is performed in the `super._update` call but not before arrays are being iterated once. If there is a length mismatch there is a high chance that the `for` loop will end up accessing bad data locations and the transaction will revert.
- Again in the `ERC1155Supply._update` function, the user input parameter `amounts` is deducted from `_totalSupply[id]` without any previous balance check, meaning that the subtraction can actually overflow, reverting the transaction. The same happens in the `increaseAllowance` function of `ERC20` and in the `safeIncreaseAllowance` of `SafeERC20`.

Following the "fail early and loudly" principle, consider including specific and informative error-handling structures to avoid unexpected failures.

Update: *#4398.*

M-03 Lack of `Context` Usage

Line 115 of `ERC20FlashMint` uses `msg.sender` instead of the `_msgSender` function to get the message's sender, even though the docstrings of the `_msgSender` function say:

> Provides information about the current execution context, including the sender of the transaction and its data. While these are generally available via msg.sender and msg.data, they should not be accessed in such a direct manner, since when dealing with meta-transactions the account sending and paying for execution may not be the actual sender (as far as an application is concerned).

This effectively prevents the use of such a contract with meta transactions (additionally not adhering to the EIP) since `msg.sender` is taken as `initiator` while it might just be the trusted forwarder of a meta transaction.

In the worst-case scenario, the `receiver` (or Borrower) implementation is the one suggested by the flash loan EIP, and in the case of a meta transaction, the transaction will revert because the `msg.sender` will not be the `receiver` itself.

Consider fixing it using the `_msgSender` function. Alternatively, consider adding a comment explaining why `msg.sender` is used instead of `_msgSender`, to avoid contradicting the implementation of this function with the documentation provided in the `Context` contract.

Update: *#4398.*

M-04 Tokens Might Get Stuck in the Contract - Phase 1

There are some places in the codebase where without any user custom implementation, the library leaves open the doors for `ERC20` tokens to get stuck inside the token contract itself.

The `ERC20Wrapper` contract has a `depositFor` function that checks whether the `sender` is `address(this)` or not but never checks whether the `account` parameter, to which `ERC20Wrapper` tokens are minted, is the same `address(this)` or not. If it happens to be `account == address(this)` then tokens are effectively stuck in the contract, as long as the user doesn't create a custom implementation with a function that is capable of transferring them out of the contract. Similarly, the `withdrawTo` function never checks the same when doing the `safeTransfer` instruction. Since there's no assumption on what `underlying` looks like, if the underlying token accepts self-transfers, an `ERC20Wrapper` `amount` of tokens will be burned, while the same amount of underlying is not transferred anywhere else. In

this case, the situation can be recovered by the use of `_recover` which will mint again the spread and bring back a 1:1 ratio between the two tokens.

Further, the `_recover` function itself allows the caller to send any stuck underlying tokens in the contract to an account sent by parameter. This amount is calculated by checking the difference between the contract's underlying amount and the wrapped token's total supply. However, if there are underlying tokens that are stuck, if someone calls the `_recover` function and provides the contract itself as a parameter (i.e., the `ERC20Wrapped` contract), then the total supply of the underlying token will have again a 1:1 ratio with the `ERC20Wrapped` contract, and those funds also will get stuck.

The same exact issue happens with `ERC20FlashMint`, where the receiver of flash loan fees is not checked to not be `address(this)`, eventually letting tokens be stuck within the contract again.

Since the library doesn't provide built-in functions to sweep tokens stuck in the contract, consider either creating a feature which is easy to plug in but that is optional or completely avoiding such edge cases by implementing proper checks. Leaving the door open for tokens to be stuck in the contract with no way to sweep them out is a common error, as we can see from USDT, DAI or many other token contracts which hold millions of dollars that are stuck there forever.

Update: *#4398.* `flashFeeReceiver` `address(this)`.

> *I* `flashFeeReceiver` `address(this)`, *.*

M-05 `ERC2771Forwarder` May Call Receiver Without Appending Sender's Address

If the `refundReceiver` in the forwarder's `executeBatch` method is set to be the receiver contract, it will result in a call that is not compliant with the EIP since it does not append the sender's address to the end of the calldata.

While the EIP specification prevents incorrect address extraction in this case, as the calldata would be shorter than 20 bytes, this protection may not be implemented by all receivers. As a result, if the forwarder is trusted by a receiver that does not apply the length rule, the receiver may extract an incorrect sender address (e.g., `address(0)`). This can result in unexpected consequences, as this address is often used as a default or burn address, and is presumed to

be an impossible sender. For example, tokens explicitly or implicitly owned by `address(0)`, or contracts with revoked ownership, may be exploited. The likelihood of this occurring, despite the multiple prerequisites, is demonstrated by the existence of this receiver vulnerability (described separately) in this codebase.

Consider making the call compliant with the EIP by appending the contract's own address to the calldata as the ERC-2771 `msg.sender`. Alternatively, consider avoiding the need to make an arbitrary destination call by utilizing WETH to make a token transfer to the refund recipient.

Update: *, #4502 06 352.*

> `ERC2771Context` .
> *-02.*

M-06 Immutable Beneficiary Security Risks and Potential Loss of Funds

The `VestingWallet` contract's beneficiary is immutable. However, during an extended vesting period, the beneficiary might need to migrate to a different account, either due to a suspected security breach of their private key or to bolster account security by shifting to a hardware wallet, a multisig, or an account-abstracted wallet.

The likelihood of such scenarios increases with longer vesting schedules. Over time, the beneficiary's security needs and the available wallet options are likely to evolve. Additionally, the unvested tokens' value could potentially grow substantially, necessitating a more robust security solution than initially available or justified.

In addition to the above, the contract does not provide a way for the beneficiary account to prove its ability to control the contract before funds are transferred into it. As a result, if the beneficiary's address is set inaccurately - be it to a contract address on another chain, an address that requires aliasing on L2, or simply an erroneous EOA - funds forwarded to the contract could be permanently lost. Although this could be attributed to a benefactor's error, the probability of such a mistake is substantial, given the benefactor and beneficiary are separate entities, and the benefactor is like to be managing multiple beneficiaries through various communication channels.

Consider modifying the contract to inherit from `Ownable2Step`, assigning initial ownership **to the benefactor** rather than the beneficiary - for instance, to `msg.sender` - and nominating

the beneficiary as the **pending owner**. At the same time, restrict access to `release` methods to `onlyOwner`, which prevents a malicious `release` from frontrunning an ownership transfer. Crucially, this also allows the benefactor to recover any funds sent to the contract if the beneficiary fails to invoke `acceptOwnership` eventually.

It's worth noting that while explicit ownership and transferability facilitate over-the-counter (OTC) trading of unvested tokens, this is also possible with the current immutable beneficiary design, given it could be an ownable contract or even a counterfactual contract if an EOA check is included.

Update: *#4508. `Ownable`, . , , . , .*

> *`beneficiary` . , `Ownable` `Ownable2Step` ' . `Ownable` 2 .*

M-07 Unnecessarily Complex and Limited Design of `customRevert` Callback

Several methods in the `Address` library (1, 2, 3, 4, 5, 6, 7) expect an argument of the `function() internal view customRevert` form. Users are thereby expected to define a distinct reverting view method, and then pass it as a function callback, similar to the procedure for `defaultRevert`.

This design has three disadvantages:

1. The "second" revert must be used to ensure that `customRevert` reverts, but it is not sufficient. For example, a `RETURN` opcode in an assembly block can bypass the rest of the Solidity code, including this second revert. Although an edge case, this will cause a failed call to not revert, violating a core intent of the library's methods.
2. The design does not support the customizability of the custom errors, as it does not allow passing parameters into the callback.
3. This interface is not a common pattern in Solidity, making it error-prone for users.

Instead, consider using `bytes memory revertData` as the input parameter for the methods. This approach offers several benefits:

- It guarantees a revert.
- It maintains the option for a simple error.
- It allows users to pass a parameterized error.
- It enables users to conform to the string-based `revert` / `require` interface if that is their preference.
- It presents a simpler, less error-prone interface for users and eliminates the need to define a special callback.
- It reduces the bytecode size of the Address library by removing `defaultRevert` and simplifying `_revert`'s implementation.

Example code:

```
error SomeError();
error RichError(uint value);

function userMethod() internal {
    bytes memory errData;

    // will look like `revert SomeError()`
    errData = abi.encodeWithSelector(SomeError.selector);

    // will look like `revert RichError(42)`
    errData = abi.encodeWithSelector(RichError.selector, (42));

    // will look like `revert("some error string")`
    errData = abi.encodeWithSignature("Error(string)", ("some error string"));

    Address.libraryMethod(errData);
}

...

function libraryMethod(bytes memory errData) internal pure {
    assembly {
        revert(add(32, errData), mload(errData))
    }
}
```

Update: *#4502* *316 30* .

> *3* *customRevert*
> . ,
> ,
> .

M-08 State Updated in Modifiers May Be Corrupted

The `ReentrancyGuard.nonReentrant`, `Initializable.initializer` and `Initializable.reinitializer` modifiers can be abruptly interrupted if a `RETURN` opcode in an assembly block is part of the wrapped method execution in `_;`.

In the case of `nonReentrant`, anything guarded by it will be permanently locked because `_status` will remain `_ENTERED`.

In the case `Initializable`, subsequent calls to `reinitializer` will be locked, and also `onlyInitializing` protection will be broken.

While a DoS due to broken `nonReentrant` and `reinitializer` may be noticed during development, and possibly fixed by an upgrade, the impact of losing `onlyInitializing` protection can be a total loss of funds and ownership of a contract. This is because a dysfunctional `onlyInitializing` can easily be missed during development and allow the contracts relying on it to become exploitable.

Consider using the `block.number` or part of its hash (either of which can be cast to be smaller than a full slot) as flags instead of `bool` values. This way, the stored values can be invalidated after loading if the current `block.number` does not match the loaded value. As a result, any state corruption will have an effect on a single block only, rather than in perpetuity. Additionally, consider adding a warning in the documentation against the usage of the `RETURN` opcode in assembly blocks in code that is expected to be used with these modifiers.

Update: , .

M-09 Function `withUpdateAt` Does Not Behave as Expected

The `Time.sol` contract provides a `Delay` type which holds a delay value that can be configured to be updated automatically at a future timepoint. The delay value may be

expressed in terms of block timestamp or block number values. The contract also provides a number of helper functions for manipulating the `Delay` type. The contract is supposed to treat block timestamp and block-number-based delays equivalently, and every helper function is implemented in two separate variations in order to handle both delay units.

However, the functions `withUpdate` and `withUpdateAt` are not equivalent. More specifically, `withUpdate` guarantees that in the case of decreasing the current delay value, it cannot be updated instantly but rather its effect timepoint adheres to a `minSetback` period. This is the expected behavior according to the `Delay` docstring. On the contrary, `withUpdateAt` immediately applies the provided effect timepoint for the new delay value.

Consider reimplementing the `withUpdateAt` function so that it behaves similarly to `withUpdate` in order to avoid mistaken expectations that could lead to misuse of the library.

Update: *#4555* *#4606*. - -

.

> `withUpdateAt` `withUpdate`.
>
> `effect=0`,
>
> " " "
>
> " " " " ", /
>
> . `effect=0`
>
> .

M-10 Proposal Execution Could Fail Due to Zero-Delayed `AccessManaged` Targets

The `GovernorTimelockAccess` contract handles proposals that may contain calls to restricted functions managed by an `AccessManager` instance (`_manager`). Upon a proposal submission, the `_manager`'s `canCallWithDelay` function is called for each target which returns a tuple `(bool allowed, uint32 delay)` denoting the respective access restrictions, if any.

The `GovernorTimelockAccess` contract ignores the boolean return variable and only considers the `delay` value to determine whether the target is managed by `_manager`. In case of a non-zero delay, the target will be scheduled during the queuing step and later on executed via the `_manager`'s `relay` function. Otherwise, the target function will be called directly by the governor, as it is considered to be a target that is not managed by `_manager`.

Note that executing through the `_manager`'s `relay` function is necessary in cases where the target is an `Ownable` contract that has transferred ownership to `_manager` (similarly for an `AccessControl` contract). For code simplicity, the current implementation executes all targets that are found to be managed by `_manager` through `relay`.

However, there is one case where `delay` is zero although the target is managed by the `_manager`. This is when the `canCallWithDelay` function returns `(true, 0)`, which means that the target is managed by `_manager` but the caller is authorized to call the target without a delay.

Consider respecting this case in the implementation to prevent this scenario from failing proposal executions.

Update: *#4591.*

> . ,
>
> `AccessManager` .

M-11 Contradictory `_cancel` Behavior

In the `GovernorTimelockAccess` contract, the `_cancel` function handles `AccessManager`-related logic to cancel previously scheduled operations. This internal function is getting called from the external `cancel` function that allows users to cancel their proposal while it is in a pending stage.

However, the internal function is attempting to cancel scheduled operations from the `AccessManager`, which due to the pending state requirement cannot exist in the first place. In the code, this is indicated through the `eta` value which would not be set until the proposal has been queued after a successful voting.

In addition, note that `_cancel` handles the canceling of succeeded proposals and it would be expected that only governance would have permission for such a sensitive action. Consider documenting the risks in case this functionality is enabled when the `cancel` restrictions of the `Governor` contract are overridden.

Furthermore, even if the logic would be overridden to allow to `_cancel` a queued proposal, then the `_cancel` function can leave governance in a corrupted state. This can happen in the event that an admin of the `AccessManager` cancels an operation directly. Then, if a user wanted to cancel the whole proposal, the call to cancel for one of the operations would fail and

revert the transaction. Hence, this semi-cancelled proposal could not be set to a canceled state and the other scheduled operations in the `AccessManager` would eventually expire.

Consider clarifying what the intention of the internal `_cancel` function is and how it fits into the overall functionality. Additionally, implement the `cancel` calls to the `AccessManager` with the necessary exception handling for operations which have already been cancelled by another party.

Update: *#4591.*

> `_cancel` ", `GovernorTimelockAccess`, , `_cancel` . .

Low Severity

L-01 Potentially Incorrect `maxFlashLoan` Amount When Using `ERC20FlashMint` And `ERC20Capped` Together

The `maxFlashLoan` function in the `ERC20FlashLoan` contract determines the maximum amount of tokens that can be flash-borrowed in a transaction. It is calculated as the difference between the maximum value of a `uint256` variable and the `totalSupply` of the token.

In addition, the `ERC20Capped` contract allows developers to set an upper limit, or cap, on the total number of tokens for an `ERC20` contract. During the minting process, the `totalSupply` of the token cannot exceed the specified `cap`, which is defined at the time of contract creation.

However, when using both the `ERC20FlashMint` contract and the `ERC20Capped` contract together, the `maxFlashLoan` function does not take the `cap` into account. As a result:

- Users who call the `maxFlashLoan` function may receive an incorrect maximum amount. If they rely on the value returned by `maxFlashLoan` and request an amount higher than the difference between the total supply and the `cap`, the transaction will fail and revert.

- When calling the `flashLoan` function, lines 109-112 allow the code to proceed even if the provided amount exceeds the cap.

Although this behavior does not pose a security risk because the `_mint` function called by `flashMint` triggers the `_update` function in the `ERC20Capped` contract, which checks that the new total supply does not exceed the `cap`, it can be confusing because the `maxFlashLoan` function does not accurately represent the true maximum amount of tokens that can be flash-borrowed.

Consider adding an internal virtual function within `maxFlashLoan` instead of relying solely on `type(uint256).max`. By doing so, if the `ERC20Capped` contract is also used, users can override the function to return `super.cap()` instead, ensuring that the maximum amount reflects the capped value.

Update: *#4398*.

> `maxFlashLoan` .

L-02 Context Contract Is Not Used

The `TransparentUpgradeableProxy` uses plain `msg.sender` instead of the `Context` contract. This makes it impossible to be used with meta transactions and is inconsistent with the rest of the contracts in the codebase.

Consider using the `Context` contract instead.

Update: *#4502* *351565* .

> `msg.sender` `_msgSender()`.

L-03 `ERC2771Forwarder` Must Not Hold Token Approvals

If `ERC2771Forwarder` is granted a token approval by the user, their tokens can be exploited. This is because an attacker can call the `execute` method to exploit the approval by forwarding a `transferFrom` request to the token contract. While the attacker address will be appended to the request, it will be ignored by the token contract, which will result in a malicious token transfer from the approving user's wallet.

Although it is unlikely that this contract will be granted any token approvals for its current functionality, given that the library contracts are typically extended with additional functionality the documentation should be clear about this risk.

Consider clarifying in the documentation that no token approvals or permits should be granted to this contract or any contract that extends it.

Update: *#4502 1 3544, 11213 5, 889 112, #4519 83 8 .*

.

L-04 Error-prone Failure Semantics of `verify` in `ERC2771Forwarder`

The `verify` view in `ERC2771Forwarder` exhibits ambiguous behavior in certain failure scenarios. It returns `false` if a request is expired or if the signer doesn't match. However, it might also revert due to the use of `ECDSA.recover`, which reverts on signature recovery errors.

This ambiguity in failure semantics can be error-prone and complicate integrations. For instance, an on-chain contract or off-chain script invoking `verify` should anticipate both a `false` return value and a potential revert. If only one of these outcomes is expected, this could lead to unforeseen consequences. An on-chain contract might revert unexpectedly if only `false` is expected. Conversely, if only a revert is anticipated, requests may be incorrectly deemed verified.

Consider using `ECDSA.tryRecover` in the `verify` view, and return `false` in all invalid scenarios.

Update: *#4502 53 4 .*

> *ECDSA.tryRecover.*

L-05 Inconsistent Solidity Version Used in `ERC1967Utils`

Most of the contracts utilize Solidity version `0.8.19`, however, the `ERC1967Utils` contract uses `0.8.20`. This discrepancy could be problematic for certain chains, especially if they do not support the `push0` opcode. This inconsistency might pass unnoticed in a project's configuration and lead to the use of an incompatible compiler version and deployment of incompatible bytecode.

Consider aligning the Solidity version in `ERC1967Utils` with the rest of the contracts to avoid potential compatibility issues.

Update: *#4489 00 5 .*
`0.8.20` ' `evmVersion`.
2 - `push0`
. ' ,
.

> *0.8.20,*
> *`ERC1967Utils` .*

L-06 Lack of Access Control and Flexibility in `VestingWallet`'s Release Methods

The release methods in the `VestingWallet` contract, `release()` and `release(address)`, lack access control, allowing anyone to invoke them. This presents a potential for malicious actions, including:

- Enabling attack vectors if the beneficiary has significant token approvals for certain contracts. If these contracts are compromised, an attacker could trigger `release(address)` and exploit any vested but unreleased tokens. In such cases, the users might consider the approvals normally safe, for instance, if they typically hold these token balances only for a short period.

- Inducing unwanted taxable events. The beneficiary may wish to time the release to their advantage in terms of taxation. A maliciously-timed release could impact the beneficiary by triggering unwanted tax liabilities.

Moreover, the transfer destination is not flexible, which might be problematic if the beneficiary's income and tax structure changes over time. This could restrict the beneficiary's ability to direct tokens to different addresses, for example, when an individual has incorporated or wants to release tokens into an entity-controlled address.

Consider restricting the release methods to be callable only by the beneficiary. Along with this access control, consider allowing the beneficiary to specify the transfer destination or nominate another beneficiary instead.

Update: , *#4508*.

L-07 Potentially Trapped ETH in `ERC2771Forwarder`

The `ERC2771Forwarder` contract is carefully designed to prevent the accidental trapping of ETH by matching incoming and outgoing ETH values and by not including a `receive` function. However, a scenario exists where the contract can still receive ETH. If the `refundReceiver` is set to the contract's own address in the `executeBatch` method, it can send ETH to itself.

Although setting the contract's own address as a `refundReceiver` can be considered a user error, this is also the case for other currently prevented potential mistakes that could result in sending ETH to the contract unintentionally.

Consider adding a check to prevent the `refundReceiver` from being set to the contract's own address, in line with the existing efforts to avoid such accidental scenarios.

Update: .

2771 ' `receive()`

.

.

L-08 Reentrancy Risk in `ERC1967Utils._setBeacon`

Within `ERC1967Utils._setBeacon`, when the `implementation()` external call is made, the assignment `StorageSlot.getAddressSlot(BEACON_SLOT).value = newBeacon;` has not yet been executed. Thus, if the `getBeacon` getter or the EIP1967 storage slot is read during this external call, it will return an inconsistent value.

Consider updating the beacon address in the storage slot prior to making any external calls. This could prevent potential inconsistencies due to reentrancy.

Update: #4502 94 .

.

L-09 Risk of Division by Zero in `VestingWallet`

In the `VestingWallet` contract, if the duration is set to zero by the deployer, the contract behaves similarly to an asset timelock for the beneficiary. However, in this case, there's a risk of division by zero in the `_vestingSchedule` function. If the duration is set to zero, `start()` and `end()` will be identical. Thus, calling `_vestingSchedule` with a `timestamp` equal to `start()` and `end()` will result in a division by zero error in the vesting formula, causing the transaction to revert for that particular block.

Consider using `>=` in the `else if` branch condition to prevent the division by zero in this scenario.

Update: #4502 9 6 9.

.

L-10 Risk of Ownership Loss Due to Single-Step Ownership Transfer in `UpgradeableBeacon` and `ProxyAdmin`

The `UpgradeableBeacon` and `ProxyAdmin` contracts utilize the `Ownable` (1, 2) contract for access control. This single-step ownership transfer method can be error-prone, potentially leading to loss of ownership.

Consider adopting the `Ownable2Step` contract instead.

Update: , .

> ' . , *ProxyAdmin* *UpgradeableBeacon* *Ownable* . , *Ownable2Step* ' - .

L-11 ERC-165 Check Is Too Permissive

ERC-165 specifies that a compliant contract must return a bool, implementing the interface `supportsInterface(bytes4) returns(bool)`. However, the check in `ERC165Checker` will return `true` for a broader set of return values, such as a `uint` that is greater than 0, or a bytes array of any nonzero size. In general, any return data of a length greater than one word, in which the first word is not 0, will cause the check to return `true`. In contrast, Solidity's `abi.decode` will revert for such values, for example `abi.decode(abi.encode(uint(2)), (bool))` will revert.

Consider restricting the check to require `returnValue == 1` to mimic Solidity's decoding behavior of a single `bool(true)` return value. Additionally, consider restricting the check of `returndatasize` to exactly 32 bytes, despite the fact that this will be stricter than Solidity's `abi.decode` behavior, to more strictly enforce the expected interface.

Update: , .

> *-165 ' I 2 . 2, , . .*

L-12 `address(0)` Is Allowed as the Initial Owner

When transferring ownership in the `Ownable` contract, the address of the `newOwner` is checked against `address(0)` in order to avoid completely renouncing the ownership by mistake. Instead, the special function `renounceOwnership` should be called for this purpose.

However, upon constructing an `Ownable` instance, the initial owner address is not checked against `address(0)`.

Consider adding a zero address check for the initial owner address upon construction, to avoid wrongly deployed instances but also to remain consistent with the rest of the contract's code.

Update: *#4531.*

L-13 Enumeration Methods Are Unnecessarily Limited

The enumeration methods of `EnumerableSet` (1, 2, 3) and `EnumerableMap` (1, 2, 3, 4, 5) only support returning the full array. The documentation states that this action can only be useful in the context of off-chain views, since it may be either too expensive to run or may even not be possible to run at all if the array is long enough to exhaust a single block's gas limit.

This is problematic because if the goal of these data structures is to provide an set and map, the goal is not accomplished as currently implemented, and this implied core functionality is not fully usable. That said, allowing paginated or sliced enumeration should be a simple and limited update to the logic of `EnumerableSet`'s `_values()`. In addition to the logic, the interfaces would need to be updated to allow passing additional arguments into the `values()` and `keys()` methods to specify either the pagination or the slicing parameters: - For pagination, a page size and page index would need to be passed, where array length and index 0 would replicate the current behavior. - For slicing, the start index and end index (or number of elements) would need to be passed, where start index 0 and array length would replicate the current behavior.

Consider implementing slicing or pagination in the keys' and values' enumeration methods to allow them to be used more flexibly.

Update: , .

> `at` `length` , .

L-14 Proposal Front-Running Protection Fails Silently

In the `Governor` contract, the `_isValidDescriptionProposer` function checks whether the description includes a `#proposal=0x` string followed by an address. As a front-running protection measure, this is to make the proposal optionally proposer dependent for the respective proposal id.

The validation function checks the proposer address string towards the end of the description and silently ignores the validation when an unexpected format is given. This means that a proposal with a simple mistake, such as a trailing whitespace, can still be front-run.

Consider making the validation more tolerant for user mistakes by parsing the whole description for the proposal query parameter and reverting if the address is badly formatted or doesn't match the proposer.

Update: , .

> . ' , , .

L-15 `TimelockController` Allows Sending ETH by Default

The `TimelockController` contract has an empty `receive` function to enable other accounts to send ETH to it. This might be necessary to further execute scheduled calls that require ETH. However, this also enables users to accidentally lose their ETH by sending it to the contract.

Consider either removing the `receive` function so that it needs to be implemented in an extended contract if it plans on handling ETH, or protecting the function through the existing proposer, executor, and canceller roles.

Update: , .

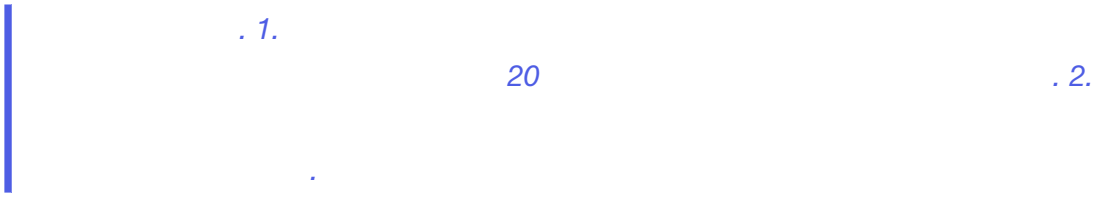
L-16 Unintuitive and Inconsistent Proposal State Timing

In the `Governor` contract, a proposal has a life-cycle of multiple states. E.g., the state can be pending or active depending on the vote start and end time.

These states are implemented such that on the vote start time point, the proposal is still pending. This behavior can be unintuitive and potentially cause confusion for applications. Consider adjusting the bound to be inclusive towards the beginning of the voting period.

In addition, on the deadline time point the proposal is still active while in the `TimelockController` contract the delay deadline time point is excluded from the delayed period. Consider adjusting the code so that a common convention is followed regarding time periods bounds throughout the codebase.

Update: , .



L-17 Overloaded Error Messages

Throughout the codebase the following instances of overloaded error messages were noted:

- In `Initializable.sol`, the error `AlreadyInitialized()` on line 146 and line 186 are not accurate as the contracts are in the process of initializing, thus not already initialized. Consider using a custom error reflecting the state that it is `StillInitializing()`.
- In `MerkleProof.sol`, the error `MerkleProofInvalidMultiproof()` is used in two different scenarios. The first appearance is when the lengths of leaves, proofs and flags violate the invariant and the second appearance is when there are unused elements left in the proof array. Consider using different and informative error messages for these two distinct cases.

Update: *#4592.*

> *1. I I . 2. . " " . , .*

Notes & Additional Information

N-01 Allowances And Approval Inconsistencies - Phase 1

The `ERC20` contract allows for self approvals, while `ERC1155` doesn't allow for that. Moreover, the `ERC1155` contract can approve an allowance to the zero address, while this is not possible in the `ERC20` contract.

Whether this inconsistent behaviour is given by following EIP guidelines or not, consider stating that in the docstrings. If there are no EIP-imposed restrictions, consider making the library consistent in how it handles approvals and allowances across all the contracts.

Update: *#4398.*

> *. - `address(0)` 1155 20.*

N-02 Gas Optimization - Phase 1

There are some areas of the codebase where either its style can be improved or changes can be made to save gas:

- The `_msgSender` function is called three times when transferring in the `ERC1155` contract. It might be worth caching its value.
- In line 67 of the `ERC1155Supply`, the `ids[i]` value is cached in a local variable, while in line 58 `ids[i]` is not. Consider making the style consistent.

Update: *#4398.*

N-03 Contracts Are Not `abstract`

With the goal of extensibility and customization, all contracts within the library, apart from some special examples, should be marked as `abstract`, so that users consciously inherit from them even if they don't include any custom implementation. However, there are still some contracts that are not marked as `abstract`, but should be. Some examples include the `ERC20` and `ERC1155Holder` contracts.

Consider reviewing the entire codebase and marking all necessary definitions as `abstract`.

Update: *#4010.*

N-04 `EIP-3156` Inconsistency

The `ERC20FlashMint` contract is an implementation of the ERC-3156 extension. Following the description of the EIP, the way in which the design was originally thought was by having an external ERC-20 token that should have been transferred to the user and then pulled back from the `receiver` plus an amount of fees.

The pull mechanism of payment is generally preferred over a push payment pattern and this is the only reason why it has been adopted by the EIP. A subtle difference comes with the flash mint variant of the EIP, represented by the in `ERC20FlashMint` contract. The difference is in that the contract itself is the ERC-20 token used, and no external tokens or contracts are used. While adopting such EIP, the intention to follow it strictly but with such subtle differences created an inconsistency.

Given the fact that there's no need to pull tokens from the receiver since the contract itself is the token contract, instead of a pull payment pattern, internal functions are used directly. This means that the need to approve an allowance beforehand and then spend such allowance is a useless design that only wastes gas. The only reason that justifies such a sub-optimal pattern is given by the fact that making a useless approval and subsequent spend allowance would allow covering either for a flash loan or a flash mint, without worrying about which one is the specific implementation of the interacting contract.

Consider either removing the need to approve and then spend the allowance or documenting why it has been maintained despite not being necessary. On a side note, consider creating a `FlashLoan` contract that adheres completely to the flash loan use case (not flash mint),

having an external token contract and a pull payment mechanism to retrieve flash-loaned funds. This new contract together with the already existing `ERC20FlashMint` should be complementary to each other, covering all use cases.

Update: , .

> *1) -3156*
> *.*
>
> *2) `flashLoan` `FlashMint` `FlashLoan` .*
>
> *, I ' . `FlashLoan` , `FlashMint` .I ,*
> *.*

N-05 Missing Interface

The `IERC20Permit` interface defines the `nonces` getter definition, while in reality, it should extend from a non-existing `INonces` interface.

Consider creating an `INonces` interface, and making `Nonces` and `IERC20Permit` extend from it.

Update: , .

> *, I -2612.*

N-06 Missing Or Incorrect Docstrings - Phase 1

Throughout the codebase, there are instances where docstrings are missing, incorrect or can benefit from a rephrase:

- Docstrings in line 71 of the `ERC1155` contract say that it is a requirement for the address to not be the zero address, but this is not enforced.

- The `uri` function of the `ERC1155` contract ignores the unnamed parameter, which might be specified in the docstrings too.
- The `_asSingletonArrays` function has an uncommented assembly block, consider clarifying each line of assembly instructions. Moreover, the function itself has no docstrings at all.
- The `_safeBatchTransferFrom` function docstrings lack the requirement for `ids` and `amounts` lengths to be equal.
- The `_mintBatch` function docstrings don't specify that `to` can't be the zero address.
- The `_burnBatch` function docstrings don't inherit the same requirements of the equivalent `_burn` function.
- The acceptances checks miss any sort of docstrings, that might need to be inherited from `IERC1155Receiver`.
- The docstrings of the `ERC1155Pausable` mention that if some don't extend from it, the contract will be unpausable. It should also be stated that the contract will not be pausable at all.
- Line 80 of the `IERC1155` has double quotes around "account".
- The `ERC1155InsufficientBalance` error definition lacks `@param` docstrings for the `tokenId` parameter.
- Line 80 of the `Address` library says that `customRevert` must be a reverting function, but this is not true, and it's not enforced.
- The `decreaseAllowance` function should specify that it doesn't protect from having the spender front-running any attempt to decrease the allowance. `increaseAllowance` and `decreaseAllowance` are described to be a safe way to avoid double-spending when using `approve`. However it does not inform that the spender can still spend the entire allowance by front-running any attempt of allowance reduction. Consider adding some clarification in the docstrings of the `decreaseAllowance` function.
- Docstrings in line 14 of `Checkpoints` say "Checkpoints.History" when it should be "Checkpoints.Trace224".
- `key` values of `Checkpoints` library should be never accepted as direct user input. If this happens anyone can disable the library by setting a `key == type(uint32).max` then any `_insert` call will revert. This should be clearly stated in the docstrings, since it is important to know when it comes to using and integrating with it.
- The `nonces` function of the `Nonces` contract should be described as the "next unused nonce" and not as the current nonce.

Consider reviewing the entire codebase for more occurrences and fixing the suggested ones.

Update: *#4398.*

N-07 Naming Suggestions - Phase 1

To favor explicitness and readability, several parts of the contracts may benefit from better naming. Consider making the following changes:

- The `ERC1155InsufficientApprovalForAll` name suggests that the allowance is not "sufficient" in quantity but the approval function doesn't take quantities into account. Consider renaming to something like `ERC1155LackOfApprovalForAll`.
- The `onERC1155Received` and `onERC1155BatchReceived` functions have a parameter named `value(s)` in the interface definition, but what is passed in when called is `amount(s)` in the actual `ERC1155` implementation. Consider making them consistent and changing `value(s)` to `amount(s)`.

Update: *#4381 #4398.*

> *. " " " ",*
> *-20 -1155 " " .*

N-08 Outdated Solidity Version

The majority of contracts use a pragma Solidity version `^0.8.19` while the `0.8.20` version is already out. Consider whether it is worth updating the version across the codebase.

Update: *, .*

> *0.8.19 0.8.20.*
> *0.8.21 .*

N-09 Uncommented Sensitive Operation

The `ERC20` and `ERC1155Supply` `_update` function relies on built-in overflow checks in Solidity to increment supply when minting new tokens. This will in general protect from overflowing.

However, it's not written anywhere in the docstrings that this is an extremely important assumption that should never change. If this was done inside an unchecked box instead, the effects might spread in many extension contracts. The main reason is that all other calculations rely on that operation to not overflow, to save gas and use `unchecked` boxes anywhere else instead.

Since such an operation has no comments at all and it is of utter importance, consider whether this should be specified in the docstrings since an eventual overflow might have a catastrophic impact even in the most basic contracts like `ERC20Capped`.

Update: *#4398.*

N-10 Unused Custom Errors

The `ERC1155InvalidOwner` custom error of the `ERC1155` contract is not used. Moreover, it is misplaced, since all custom errors are defined within the `contracts/interfaces/draft-IERC6093.sol` file. The same happens with the `ERC1155InvalidApprover` custom error, which is not used anywhere in the codebase.

Consider removing unused custom errors and to consolidate all custom error definitions within the same file.

Update: *#4261.*

> *`ERC1155InvalidOwner` . `ERC1155InvalidApprover` ,*
> */ .*

N-11 Unused or Duplicated Imports And Extensions - Phase 1

In the codebase, there are cases in which contracts might have a shorter inheritance chain or where imports are either unused or duplicated.

- In `Checkpoints.sol` the import `SafeCast` is unused and could be removed.
- The `ERC1155` contract extends from `IERC1155` and `IERC1155MetadataURI` but the latter already extends from the former. Similarly, some imports are already present in parent contracts.

Consider removing unused or duplicated imports and/or extensions to improve the overall clarity and readability of the codebase.

Update: *#4398.* ,

.

-1155 ,

`IERC1155MetadataURI` `IERC1155`.

N-12 Outdated Version in Docstrings

Many contracts don't have the docstrings updated with the latest version. An example is the `ERC1155` contract that mentions `v4.9` in the docstrings titles.

Consider updating all of the codebase's docstrings to reflect the latest version of the contracts that have been modified in the latest release.

Update: , .

.

N-13 Event Definition Improvement Suggestions

The `AdminChanged` event of `ERC1967Utils` does not index its parameters, while the `Upgraded` and `BeaconUpgraded` events do not emit the old implementation's values.

Consider indexing event parameters and emitting old values whenever a change occurs in the state.

Update: , .

-1967 . ' .

N-14 Code Style Suggestions - Phase 1

Throughout the codebase, many functions use the nested conditionals coding style, which affects readability. For instance:

- In `TransparentUpgradeableProxy`, the `_fallback` function
- In `Address`, the `verifyCallResultFromTarget` function
- In `ERC2771Forwarder`, the `_execute` function

  Consider refactoring the aforementioned functions following the guard clauses technique. An example is shown below for the aforementioned `_execute` method.

[redacted]

```
  boo
internal vi
  // note: coul
ata request, boo
```

[redacted]

N-15 Inadequate Documentation for Reverting Payable Upgrades with Empty Data

Several upgrade methods within the contracts are payable, including the `ProxyAdmin` and its target in `TransparentUpgradeableProxy`, the `UUPSUpgradeable`, and the `ERC1967Proxy` .

However, these methods will revert in the `ERC1967Utils.upgradeToAndCall` if the `data` parameter is empty, yet `msg.value` is not zero. This occurs because even if the implementation can receive ETH, this cannot be guaranteed, since it will not be called due to the empty `data` input.

Consider adding inline documentation that explains why this combination of inputs is unsupported. Additionally, provide guidance on the best approach a project can use to support these inputs if they are required for any reason.

Update: *#4382* *121 5* .

> *`ERC1967Utils.upgradeToAndCall`*
> *`ERC1967Utils.upgradeBeaconToAndCall`* .

N-16 Incompatibility of `VestingWallet` with Rebasing Tokens

The `VestingWallet` contract may not function as expected with rebasing tokens. In such tokens, the token's balance and total supply can change without any interaction with the `VestingWallet` contract, due to the rebasing mechanism.

Since the `_released` and `_erc20released` variables remain static between their updates (1, 2), an external change in the contract's token balance due to a rebase, can result in the vesting schedule speeding up or slowing down, contrary to the defined `durationSeconds`.

Consider documenting this incompatibility to ensure users are aware that this contract may not function correctly with ERC-20 tokens that exhibit atypical balance behaviors, such as rebasing tokens.

Update: *#4502* *6689060*.

> .

N-17 Lack of Event Emission - Phase 2

Throughout the codebase there are some state changes and calls that are not accompanied by event emission:

- The `ProxyAdmin` contract does not emit an event during the `upgradeAndCall` function execution. Although the proxy contract will emit an `Upgraded` event, it will do so in its own context. Emitting a separate upgrade event in the context of the admin contract would simplify off-chain upgrade tracking, especially in systems where a single admin controls multiple proxies.
- The `UpgradeableBeacon`'s constructor updates the implementation without emitting an event. This happens because the `Upgraded` event is not emitted during the execution of the internal `_setImplementation` function. Consider relocating the event emission to the `_setImplementation` function to ensure it is triggered in both cases.

Consider ensuring events are emitted in these cases.

Update: *#4502 687 805.*

> `Upgraded`

N-18 Lack of Inclusion of a "Vesting Cliff" Feature

A "vesting cliff" is a prevalent feature in vesting schedules. It defines a date before which no tokens can be released, despite the linear progression of the vesting schedule prior to that date. Even though projects can extend the contract to include this functionality, given its prevalence in vesting schedules, it would be beneficial to incorporate it into the default contract. This addition would help reduce potential implementation errors and enhance the utility of the default contract.

Consider adding a `cliffTimestamp` input to the constructor. This timestamp could be used in a `cliff()` view, whose value should not exceed the `end()` time. In the `_vestingSchedule` method, if the `timestamp` is earlier than the `cliff()`, the `totalAllocation` should be set to 0.

Update: , .

.
.

N-19 Missing Or Incorrect Docstrings - Phase 2

Throughout the codebase, there are instances where docstrings are missing, incorrect or can benefit from a rephrase:

- In the ERC1967Utils library, the comments for the internal constants IMPLEMENTATION_SLOT, ADMIN_SLOT, and BEACON_SLOT mention that they are "validated in the constructor". However, no constructor occurrence could be found where the aforementioned internal constants are validated.
- In the IBeacon interface, the following comment assumes that the `BeaconProxy` will check that the implementation's address is a contract. However, this is not checked in `BeaconProxy` but rather in UpgradeableBeacon while setting the implementation.
- The definition's syntax of the BEACON_SLOT differs from the definitions' syntaxes of IMPLEMENTATION_SLOT and ADMIN_SLOT.
- Throughout the scope (and codebase), events that refer to standard EIP events are mentioned as "Emits an {IERCxxx-Eventname} event" in the comments. In some other places, standard EIP events are mentioned as "Emits an {Eventname}" instead. For example, in the comments for upgradeToAndCall and upgradeToAndCallUUPS, the declaration of the `Upgraded` event is not consistent.
- The `UUPSUpgradeable` compatibility check should specify in the docstrings that it is merely a sanity check and that more robust rollback tests are possible but not implemented because of increased complexity, code size and gas consumption.
- The `VestingWallet` docstrings should specify that the beneficiary is able to modulate the behavior of the linear vesting into a curve by re-depositing into the vesting contract the amount released for both ETH and tokens. Moreover, it should prove the calculations starting from an invariant. An example:

```
/**
 * Suggested documentation for `releasable()`
 *
 * The contract's invariant is `total-withdrawal / total-deposit <= vesting-ratio`
where:
```

```
 *    - `vesting-ratio` is 0 before vesting period start, and during the vesting period
 *    is linear in the vesting duration (to a maximum of 100%). For example 50%
 *    ratio after 50% of duration.
 *    - `total-deposits` is the sum of the contract's balance (regardless of who
supplied it and when),
 *    and the already `released` funds (because the release method is the only way
 *    that funds can leave this contract).
 *    - `total-withdrawal` is the sum of already released funds (stored in `released`),
 *    plus the currently `releasable`. Again, because the release method is the only
way
 *    that funds can leave this contract.
 *
 * From the above we can translate the invariant to:
 *    `(released + releasable) / (released + contract-balance) <= vesting-ratio`.
 * Extracting maximum `releasable` (assuming equality):
 *    `releasable = (contract-balance + released) * vesting-ratio - released`
 *
 * We can call the first term on the right side `vested-amount` and calculate it
separately
 * in `vestedAmount()` to make it available in a separate view.
 * /
```

- The `equal` function of `Strings` should specify that the function is not able to distinguish same-length input hash collisions.
- Functions that do `delegateCall`s in the `Address` library should have a warning stating that the call potentially modifies the state of the caller contract.
- The `_checkNonPayable` misses useful docstrings.
- `being impossible` in `ProxyAdmin`'s and `UUPSUpgradeable`'s docstrings should be `making it impossible`.

Consider reviewing the entire codebase for more occurrences and fixing the suggested ones.

Update: *#4502* *60329* .

N-20 Naming Suggestions - Phase 2

Throughout the codebase, there are naming choices that impact readability or are inconsistent. Some suggested improvements include:

- `implementation` is the generally adopted name for the logic contract behind a proxy. However, in the `ERC1967Proxy` and `TransparentUpgradeableProxy` constructors it is called `_logic` instead. Consider always using `implementation`.
- The constant `_SYMBOLS` can refer to any symbol. Consider renaming it to `_HEX_DIGITS` to be more precise.
- The `UpgradeableBeacon` contract is not per se upgradeable, and instead administers upgradeability for `BeaconProxy` contracts. Consider mimicking the `ProxyAdmin` contract which administers upgradeability for `TransparentUpgradeableProxy` and use `BeaconProxyAdmin` instead. Alternatively, use `ImplementationBeacon` to describe the functionality directly.
- `alive` is an atypical name and is confusing. Some possible alternatives include `expired` (flipped value), `nonExpired`, or `beforeDeadline`.
- `requireValidRequest` is confusing since it can imply that invalid calls are processed despite being invalid. Consider flipping the value and using `skipInvalidRequests`.
- Consider incorporating a clearer distinction between the variables and methods in `VestingWallet` that are duplicated between ETH and ERC-20 cases by suffixing them with `ETH`, and `ERC20` or `Token`.

Update: *#4502* *9963*.

N-21 Trusted Forwarder Address Lacks External Visibility

`ERC2771Context` stores the trusted forwarder address as an immutable variable and does not emit an event. As a consequence, querying for the address would be difficult off-chain and impossible on-chain.

Consider emitting an event when setting the forwarder address, and providing an external view to access it.

Update: *#4502* *1164_51*.

> , -
> .

N-22 Unused Named Return Variables - Phase 2

The `impl` named return variable of the `ERC1967Proxy` is unused. Similarly, in `ERC2771Context` the `sender` variable of the `_msgSender` function is unused in the `else` branch.

In both cases, consider either removing the named parameter or making use of it.

Update: *#4502* *58546_1*.

> `_impl` . `sender` '
> .

N-23 Code Style Suggestions - Phase 3

The check in `Initializable.initializer` is very complex and difficult to read, which makes it unnecessarily hard to maintain and audit. The code can be improved by documenting the logic of the check and splitting the conditional logic into multiple temporary variables that are appropriately named. For example, consider refactoring in the style of the code below:

```
// first call is allowed
bool firstCall = (isTopLevelCall && _initialized == 0);
// calls at construction time are allowed to allow multiple constructors to run their
inits.
// the version check ensures the version number cannot be reduced
// in the case version > 1 was already set by `reinitializer` or `_disableInitializers`
bool callDuringConstruction = (address(this).code.length == 0 && _initialized == 1);
if (!firstCall && !callDuringConstruction) {
    revert AlreadyInitialized();
}
```

Update: *#4576*.

> .

N-24 Some ERC-721 Features Might Be Atomically Reset

The `ERC-721` token has many extensions and among them, there are the royalties and the URI storage ones. If a particular `tokenId` is first burned and then re-minted, that specific token will have all of these features reset, the custom URI and royalties information but also the specific approvals.

Consider whether this should be documented so that users know the potential mechanisms involved.

Notice that `ERC721Wrapper` offers a `withdrawTo` and `depositFor` flow that can be called within the same transaction and will effectively burn and re-mint the same wrapped `tokenId` achieving the mentioned reset.

Consider documenting that the wrapper can be used in an atomic way, especially since the receiver of any wrapped token can be a contract on itself and re-enter the wrapper contract on the `onERC721Received` hook.

Update: *#4561.*

> ' ,
> , ' .

N-25 Missing or Incorrect Docstrings - Phase 3

Throughout the codebase, there are instances where docstrings are missing, incorrect or can benefit from a rephrase:

- The `burn` function docstrings of the `ERC721Burnable` contract should clarify why the internal `_burn` function is not used and make clear when it should be used.
- The `transferFrom` function of the `ERC721` contract that refers to the `_isAuthorized` function as "_isApprove". Similarly for the comment for the `_update` function of the `ERC721Burnable` contract.
- `hash` should be `messageHash` in `MessageHashUtils` contract.
- `ECDSA.tryRecover` return values description is out of date.
- In the `ERC721Consecutive` contract some advisory docstrings refer to the `_beforeTokenTransfer` and `_afterTokenTransfer` hooks which have been abandoned in the latest version.

- The `WARNING` docstring for the `_checkAuthorized` function in the `ERC721` contract unclearly suggests that the `_isAuthorized` function checks whether the `owner` address is the actual token owner, while this is not true.
- The onlyRole and _checkRole docstrings in `AccessControl` describe the regular expression format of the previous revert reason string, which has been replaced with a custom error.
- The documentation of `Initializable`'s `initializer` is misleading since it mentions that the function protected by it can only be invoked once, or if nested within the constructor. However, the methods protected by it can be invoked any number of times during construction, and do not have to be nested. An example contract below is supported:

```
contract MultipleInits is Initializable {
  constructor() initializer {
      initialize();
      initialize();
  }
  function initialize() public initializer {}
```

It appears that this is a known edge case since the contract documentation warns about multiple calls to `initializer`-protected methods. Consider clarifying the method documentation to prevent confusion regarding the allowed use cases and the lack of protection from multiple initializations.

Consider reviewing the entire codebase for more occurrences and fixing the suggested ones.

Update: *#4581.* *burn* *ERC721Burnable* *_burn* .

N-26 Lack of Event Emission - Phase 3

In the `ERC2981` contract, used within `ERC721Royalty`, there are no events emitted every time a custom or default royalty's information is being reset.

Consider emitting events every time a sensitive storage action is performed.

Update: , .

> *-2981,* . '
>
> .

N-27 Repeated Code

In the `ERC721` contract there are two places in which code can be optimized and reused with little refactoring.

- The `ownerOf` function could make use of `_requireMinted` logic to use the same logic.
- The `auth` parameter is checked to not be the zero address two times, one inside the `_update` function and one inside the `_isAuthorized` function, called within `_update` execution exclusively. If the `_isAuthorized` function is meant to be triggered exclusively within the `_update` one, consider removing the second check.

Consider consolidating the logic in less code and reusing the same pattern as much as possible.

Update: *#4566.*

> *721 , `_requireMinted` `_requireOwned`. `_update` . `_isAuthorized` , `_update`. `spender` , .*

N-28 Hardcoded Magic Constant

In the `ERC721URIStorage` contract, there is the `0x49064906` hardcoded value.

Consider adding a comment specifying where the hardcoded magic constant comes from and defining it in a constant variable instead.

Update: *#4560.*

N-29 `_setTokenURI` Does Not Allow Setting URI for Non-Existing Tokens

The `_setTokenURI` function of the `ERC721URIStorage` contract doesn't allow setting the URI for non-existent `tokenId`.

This prohibits using this contract with a common use case in which `tokenId` metadata are set even before minting the corresponding token.

Consider lifting this restriction to not limit the library's flexibility.

Update: *#4559.*

N-30 Default Handling Contract of the `DEFAULT_ADMIN_ROLE` Is Complex

The `AccessControl` contract is the basic building block for constructing role-based access control mechanisms. Each role may have its own administrator entity. However, the administrator account of the `DEFAULT_ADMIN_ROLE` serves as the default administrator for all roles for which a specific administrator is not defined. Thus the `DEFAULT_ADMIN_ROLE` comes with the great power that handles other roles, essentially granting or revoking access from members.

The `AccessControl` contract makes no assumptions and provides no special rules for the owner of the `DEFAULT_ADMIN_ROLE`. In essence, users are encouraged to use the `AccessControlDefaultAdminRules` contract as an extension to the basic `AccessControl`, which defines timelock-like procedures for the sensitive operations of the default admin. However, the advanced security of the `AccessControlDefaultAdminRules` contract comes with a certain level of complexity and could be erroneously used or modified by non-sophisticated development teams.

To assist developers with the somewhat complex management of the `DEFAULT_ADMIN_ROLE`, consider providing more extensive documentation for the `AccessControlDefaltAdminRules` contract, explaining the purpose and mechanics of the two delay mechanisms, along with example use cases. Also consider expanding the existing docstring of the `AccessControl` contract to provide more specifics about the advanced security guarantees provided by `AccessControlDefaultAdminRules`, such as the delay mechanisms, to help development teams choose which contract is most suitable for their projects.

Update: , .

N-31 Unused Named Return Variables - Phase 3

The named return variable `digest` is unused in `MessageHashUtils`'s `toEthSignedMessageHash(bytes memory message)` and

N-34 Missing or Incorrect Docstrings - Phase 4

Throughout the codebase, there are instances where docstrings are missing, incorrect or can benefit from a rephrase:

In `AccessManaged.sol`:

- The `_checkCanCall` function contains a confusingly named variable `allowed`, which is not purely a yes/no indicator of whether the caller is allowed to call the target function. A call may still be able to execute even if `!allowed` is true. Consider adding a comment that clarifies this behavior.

In `AccessManager.sol`:

- The comments for the `Access` data structure refer to an `onlyGroup` modifier which is not present in the codebase.
- The `_canCallExtended` function returns a `(bool, uint32)` tuple but has no documentation that explains how to interpret these values, and determining their meaning is not trivial.
- The `_getAdminRestrictions` function returns a `(bool, uint64, uint32)` tuple but has no documentation that explains how to interpret these values, and determining their meaning is not trivial.
- There is a stale docstring which refers to three possible modes `(open, custom, closed)` while no such categorization is implemented.
- The docstring of the `cancel` function should mention that in addition to the proposer and guardian, a global administrator is also allowed to call it.
- The `renounceGroup`, `revokeGroup`, and `_revokeGroup` docstrings all say "Emits a {GroupRevoked} event" instead of "May emit". If the targeted account is not a member of the group, the shared function `_revokeGroup` will return before emitting the `GroupRevoked` event.
- The docstring of `setClassFunctionGroup` says it emits a "FunctionAllowedGroupUpdated" event. This should be changed to "ClassFunctionGroupUpdated".
- The docstring of `_setClassFunctionGroup` says this function is the internal version of "setFunctionAllowedGroup". This should be changed to "setClassFunctionGroup".
- The docstring of `_setClassFunctioNGroup` says it emits a "FunctionAllowedGroupUpdated" event. THis should be changed to "ClassFunctionGroupUpdated".

- The docstring of `setClassAdminDelay` says it emits a "FunctionAllowedGroupUpdated" event. This should be changed to "ClassAdminDelayUpdated".
- The `_checkAuthorized` function contains a confusingly named variable `allowed`, which is not purely a yes/no indicator of whether or not the call is authorized. A call may still be allowed to execute even if `!allowed` is true. Consider adding a comment that clarifies this behavior.
- The `AccessMode` and `Class` structs are undocumented.

In `EnumerableMap.sol`:

- An isolated comment refers to a non-existent data structure `Uint256ToAddressMap`, which has been renamed to `UintToAddressMap`.

In `IAccessManaged.sol`:

- No documentation is present in this file for `IAccessManaged` interface and its functions, events, and errors.

In `IAccessManager.sol`:

- Except for three events, no other documentation is present in this file for the `IAccessManager` interface and its functions, events, and errors.

In `Time.sol`:

- The docstring for the type `Delay` describes it as 128 bits long, but its size is `uint112`.
- The docstring for the type `Delay` provides a visual representation of the packed values inside the `Delay` type but puts them in the wrong order: the `current value` and `pending value` labels should be swapped so that `current value` occupies the least significant `CCCCCCCC` bits in the diagram.

Consider reviewing the entire codebase for more occurrences of missing and incorrect documentation, and fixing the suggested ones.

Update: *#4586, #4581. `IAccessManaged` `IAccessManager` .*

N-35 Refactor `AccessManager` Data Structures to Reduce Design Complexity

The `AccessManager` contract organizes the access rights per each managed contract's function selector. For this purpose, the following mappings and structs are used:

- `_contractMode` mapping: Given a target contract address, returns an `AccessMode` struct. `AccessMode` contains a `classId` and a boolean value denoting whether the contract is "closed" or not.
- `_classes` mapping: Given a class's id, returns a `Class` struct. `Class` contains a mapping with the allowed group (`groupId`) per function selector and an `adminDelay` value, which is a delay applied on some class-level administrative operations.
- `_groups` mapping: Given a member group's id, returns a `Group` struct. `Group` contains a mapping with the access details per member along with some other data.

Essentially, there are two intermediary structs (`AccessMode` and `Class`) involved with linking a target contract to its member groups per function selector. While this distinction could help batch some update operations for similar target contracts which share the same class of access rules, it is unnecessary for the most common use cases. At the same time, this distinction increases the conceptual complexity of the already complex `AccessManager` contract.

Consider consolidating the `AccessMode` and `Class` structs into a single data structure that eliminates the intermediate `Class` concept, in order to simplify the design and reduce the conceptual complexity in regards to the configuration of the access management rules.

Update: *#4562.*

> *. `AccessMode` `TargetConfig`, `getContractClass` `getTargetClass`*
>
> " " .

N-36 Unused Variables

In the `AccessManager` contract, the `_adminDelays` variable is declared but never used.

Consider removing the unused variable for clarity and readability.

Update: *#4565.*

N-37 Inconsistent Use of Named Return Variables - Phase 4

Almost all functions in `IAccessManager` that return a value do not use named return variables, but the `canCall` and `getContractClass` functions do use named values. The `canCall` function in the `IAuthority` interface also returns a named value.

While inconsistent with the rest of the `AccessManager` codebase, these named return values still provide valuable information to the reader. Consider removing the named values but adding documentation that provides the same information in the functions' docstrings.

Update: , .

.

N-38 Missing Zero Address Check in `AccessManager` Constructor

The `AccessManager` constructor's job is to grant an `initialAdmin` address access to the `ADMIN_GROUP`. However there is no check that ensures `initialAdmin` is not `address(0)`, in which case the contract would be deployed without an admin role. Attempting to correct this after the fact by calling the public `grantGroup` function would fail because the function is protected by the `onlyAuthorized` modifier.

Consider checking that the initial administrator address is not `address(0)` upon the contract's construction, in order to avoid the accidental deployment of contracts that cannot be used.

Update: *#4570*.

N-39 Use of Custom Errors

The `AccessManager` contract uses custom errors in nearly all cases. In contrast, the `consumeScheduledOp` function uses a `require` statement with no error string for reverting.

To improve the consistency of the codebase, consider reverting with a custom error instead of using `require`.

Update: *#4575.*

> *. , 4*
> *, .*

N-40 Code Style Suggestions - Phase 4

In the `AccessManager` contract, the `grantGroup` function has dual-purpose functionality: it handles granting access to a new member of a group, and also updating an existing member's `delay` value. In both cases, a `GroupGranted` event is emitted, even if the member has already been granted access to the group.

To favor explicitness and simplify the code, consider adding a separate function to handle the case of updating a member's `delay` value. In addition, consider emitting a separate event for such updates.

Update: *#4569.*

> *. `GroupGranted`*
> *, `since` .I `GroupGranted`*
> *, .*
> *, .*

N-41 TODO Comments

The `minSetback function` in the `AccessManager` contract has a TODO comment that states the setback value should be changed from 0 days to 1 day. Additionally, lines 122-123 in the `canCall` docstring appear to be a TODO comment without a TODO tag.

Consider tracking these changes in the project's issue backlog.

Update: *#4557.*

> *5 , `grantGroup`*
> *0 .*

N-42 Gas Optimization - Phase 4

In `AccessManager`'s `setClassFunctionGroup` function, the length of the `selectors` array being iterated could be saved in a local variable that replaces `selectors.length` in the `for` loop.

Consider making the above change to reduce gas consumption.

Update: , .

N-43 Typographical Errors - Phase 4

Consider addressing the following typographical errors:

In `AccessManager.sol`:

- Line 48: Remove the extra space between "danger" and "associated".
- Line 59: "This structures fit" should be "This structure fits".
- Line 62: "are not available" should be "is not available" (or "group permission" should be "group permissions").
- Line 73: "grand" should be "grant".
- Line 95: "transcient" should be "transient".
- Line 126" "contract" should be "contracts".
- Line 127: "call" should be "calls".
- Line 208: "operation" should be "operations".
- Line 208: "require" should be "requires".
- Line 585: "restriction to that apply" should be "restrictions that apply".

In `Time.sol`:

- Line 47: "so guarantees" should be "some guarantees".
- Line 49: "is the delay" should be "if the delay".
- Line 116: "after at a timepoint" should be "after a timepoint".

Update: *#4571*.

N-44 Mismatch Between Contract and Interface

The `getSchedule` function in `AccessManager` is defined as a `view` function in the implementation, but in the `IAccessManager` interface, the `view` keyword is missing.

To avoid confusion, consider adding the `view` keyword to the `getSchedule` function's definition in `IAccessManager`.

Update: *#4558.*

> *`view` `getSchedule` . , `CALL` `STATICCALL`.*

N-45 Naming Suggestions - Phase 4

To favor explicitness and readability, several parts of the contracts may benefit from better naming. Consider making the following changes:

- The `EnumerableSet` contract uses the word "index(es)" to name both indexes and indexes incremented by one, which hinders the understanding of the code. Consider renaming indexes incremented by one to "position".
- The `EnumerableSet` contract refers to its elements as "value(s)". The word "value", however, is generic and creates unnecessary confusion especially when `EnumerableMap` comes into play (where the word "value" refers to the value in a key-value pair). Consider using more specific words for set elements, such as "element" or "member".
- The `AccessManager` contract uses the word `delay` to name both the execution delay of a group's member as well as the grant delay of a group. Renaming these two variables to indicate their specific purpose would improve the contract's readability.
- In the `AccessManaged` contract, the function `_checkCanCall` makes an external call to the `AuthorityUtils` contract and names the boolean return variable "allowed". However, even if `allowed` is `false`, this does not necessarily mean that the caller is unauthorized, but rather that the operation falls under a delay. In this case, if the operation has been scheduled and the delay duration has been completed then the execution proceeds normally, following an unintuitive flow. Consider renaming the variable "allowed" to a word that accurately denotes its content, such as "immediatelyAllowed".

Update: *#4577, #4562.*

> *I `AccessManager.canCall` allowed immediate.*
> *I `EnumerableSet` , " "*
> *(++,).*

N-46 Missing or Incorrect Docstrings - Phase 5

Throughout the codebase, there are instances where docstrings are missing, incorrect, or can benefit from a rephrase:

In `Governor.sol`:

- The docstring for the `_governanceCall` variable refers to `_beforeExecute` and `_afterExecute` functions which do not exist in the codebase anymore.
- The docstring for the `_queueOperations` function refers to the `_queue` function which does not exist in the codebase.
- "Any asset sent to the {Governor} will be inaccessible" could mention that assets can be recovered through the relay function of the `Governor` contract.
- The comment "ProposalCore is just one slot. We can load it from storage to stack with a single sload" is wrong, as the struct actually takes two slots.

In `TimelockController.sol`:

- The docstring for the `isOperation` function refers to a "Pending" state, which does not exist. It should refer to the `Waiting` state instead.

In `GovernorTimelockControl.sol`:

- There is a `WARNING` docstring pointing out some risks relevant to the sensitive access roles of the `TimelockController` contract. However, the list of risks is neither accurate nor complete. More specifically, the `relay` function of the `Governor` contract is in fact secure against malicious role holders. Furthermore, apart from the risk of malicious cancelers causing a DoS, it is also possible that malicious proposers and executors create and execute arbitrary proposals bypassing the voting procedure. Consider listing this risk as well. Consider also documenting that the `GovernorTimelockControl` module can be used in two different secure ways:
  - `TimelockController` is set as the owner of the target contracts. Then, in order to mitigate the risks mentioned above, a secure setup of `TimelockController` is needed where the governor contract is the only admin and proposer.

    - `GovernorTimelockControl` is set as the owner of the target contracts. Then the proposals should be structured in a way that all its actions are forwarded by the `relay` function.

In `GovernorTimelockAccess.sol`:

- Consider documenting the fact that scheduling the same call is prevented by the `AccessManager`, while as a workaround it is recommended to append extra calldata, which will function as a salt.
- Consider correcting the docstring of the `_detectExecutionRequirements` function, which appears to be outdated.

In `GovernorVotesQuorumFraction.sol`:

- "If history is empty, fallback to old storage" is outdated because the fallback mechanism has been removed.

In `GovernorSettings.sol`:

- The comment "voting period must be at least one block long" in the `GovernorSettings` contract is not accurate, since the period can also be in seconds.

In `DoubleEndedQueue.sol`:

- The docstring for the `Bytes32Deque` struct refers to the use of signed integers for indices, but the queue now uses unsigned values.
- The docstring for the `Byres32Deque` struct states that indices lie in the range `[begin, end)`, but due to intentional overflow/underflow that allows indices to wrap around, `_begin` can be larger than `_end` (Example: push 1 element to the front of an empty queue).
- For consistency, the docstring for the `pushBack` function should contain the following note: "Reverts with `QueueFull` if the queue is full".
- For consistency, the docstring for the `pushFront` function should contain the following note: "Reverts with `QueueFull` if the queue is full".

In `MerkleProof.sol`:

- The `_hashPair` and `_efficientHash` functions are undocumented.
- The `_efficientHash` function contains undocumented assembly code.

In `Initializable.sol`:

- The `reinitializer` modifier's docstring states: The comment "WARNING: setting the version to 255 will prevent any future reinitialization" is incorrect. The version value used to disable reinitialization has been changed from `255` to `2^64 - 1`.

Consider reviewing the entire codebase for more occurrences of missing and incorrect documentation, and fixing the suggested ones.

Update: *#4601.*

N-47 Gas Optimizations - Phase 5

In the `MerkleProof` contract, the `processProof` and `processProofCalldata` functions both access `proof.length` inside their respective `for` loops.

Consider saving this length value in a local variable prior to entering each loop.

Update:

N-48 Inconsistent Use of Named Return Variables - Phase 5

With the exception of `processMultiProof`, `processMultiProofCalldata`, and `_efficientHash`, the other functions in the `MerkleProof` contract do not use named return variables.

While this approach to naming is inconsistent, these named return values still provide valuable information to the reader. Consider removing the named values but adding documentation that provides the same information in the docstrings of these functions.

Update:

N-49 Unused Import - Phase 5

Within the `GovernorTimelockCompound` contract, the `IERC165` import is not used.

Consider removing unused imports to improve the overall clarity and readability of the codebase.

Update: *#4590.*

N-50 Inconsistent Solidity Version Used in `GovernorStorage`

One of the 5.0 release changes is bumping the Solidity version to `^0.8.20`. This change has been applied to all files except for `GovernorStorage`, which has version `^0.8.19` but extends a `^0.8.20`-versioned `Governor` contract anyways.

Consider adapting the Solidity version to be consistent with the rest of the codebase.

Update: *#4589.*

N-51 Long Comment Lines

The style guide of the codebase appears to foresee a maximum line-width of 120 characters. However, this is not to fully adhered to. For instance, see the comment on the `GovernorStorage` contract.

Consider adhering to the ruler of 120 characters for the code to be more readable in split-screen or diff-views.

Update: *#4600.*

N-52 Unused Named Return Variables - Phase 5

In the contract `GovernornCountingSimple` the named return variables `againstVotes`, `forVotes` and `abstainVotes` are unused in the `proposalVotes` function.

For consistency with the rest of the codebase, consider assigning values to these return variables instead of explicitly returning. Alternatively, consider removing the named variables from the signatures of the methods in these cases.

Update: , .

N-53 Typographical Errors - Phase 5

Consider addressing the following typographical errors:

- Throughout the codebase, there is an inconsistency between using the word "Canceled" and "Cancelled". Consider adhering to American or British English.

In `Governor.sol`:

- Line 19: "though" should be "through"

In `TimelockController.sol`:

- Line 167: "correspond" should be "corresponds"

In `GovernorTimelockControl.sol`:

- Line 114: "as already been queued" should be "has already been queued"
- Line 38: "where the nonce is that which we get back from the manager" should be "where the nonce is received from the manager"
- Line 21: "powers that they must be trusted" should be "powers that must be trusted"
- Line 114: "if it as already been queued" should be "if it has already been queued"

In `Nonces.sol`:

- Line 16: "an the" should be "the" .

In `MerkleProof.sol`:

- Line 16, Line 17, Line 69, Line 72, Line 86, Line 103, Line 115, Line 161: "merkle" should be "Merkle"

Update: *#4595*.

N-54 Naming Suggestions - Phase 5

To favor explicitness and readability, several parts of the contracts may benefit from better naming. Consider making the following changes:

- In `Votes.sol`, the `delegates` function only returns a single delegate address, and is similarly named to the `delegate` function. Consider renaming `delegates` to `getDelegate`.
- In `Votes.sol`, an address that delegates its voting units is often referred to as an account and an address receiving delegated votes is often referred to as a delegatee. This naming consistency is also observed in the local variables for instance in the delegate function. However there are some instances where a delegatee is referred to as an account, for instance in getVotes , getPastVotes functions. Consider adopting this naming convention consistently throughout the contract for clarity and readability.
- Throughout the codebase state variables have a leading underscore, while function parameters have no underscores. Consider sticking to this format for the `name_` parameter of the `Governor` constructor.
- The `ProposalCore` struct contains three variables that handle timepoints, namely `voteStart`, `voteDuration` and `eta`. While the time unit of the first two is expected to be decided by the `clock()` implementation, the `eta` value follows the block timestamp as happens in the `GovernorTimelockControl` and `GovernorTimelockAccess` extensions. As the timepoints are generally expected to follow the same units within a module it is possible that the users misinterpret the value of the timepoints. Consider renaming `eta` to `etaSeconds` or something similar to underline the possible inconsistency and avoid false expectations.
- The `TimelockController` implements the roles proposer, executor, and canceller. The proposer role can be confusing in the context of governance contracts, which involves making actual proposals. However, in the context of the `TimelockController`, the proposer role is able to schedule calls, which could be a successful proposal. Hence, calling the role scheduler would be more fitting.
- Throughout the `TimelockController` contract, a single call's calldata is called `data`, while a batch-call's calldata is called `payloads`. Consider adhering to `payload(s)` to clarify that it is referring to same information.
- In the `ProposalCreated` event the parameters `voteStart` and `voteEnd` indicate the starting and ending timepoints of the voting period. However, the getter functions for these values are named after `proposalSnapshot` and `proposalDeadline` respectively. Consider adopting a consistent naming to avoid confusion.

Update: *#4599.*

> *ProposalCore.eta etaSeconds.*
>
> *4.*
>
> *.*
>
> *.*

N-55 Extraneous Code

The following cases of extraneous code have been identified:

- In the `Governor` contract, the validity of performing a specific action is usually checked through the `_validateStateBitmap` function. However, this functionality is duplicated within the `_castVote` function, instead of utilizing the `_validateStateBitmap` function.
- In the `Nonces` contract, the `_useCheckedNonce` function returns (assuming no revert) the `current` nonce, which will always be equal to the input `nonce`. This eliminates the need for the caller to ever check the return value. Consider removing the return value.

Consider addressing the above cases as suggested.

Update: *#4588.*

Client-Reported

CR-01 Inconsistent Use of Hooks

The `ERC20` and `ERC1155` contracts have been refactored to delete the `_beforeTokenTransfer` and `_afterTokenTransfer` hooks in favour of a more robust `_update` function. However, the `_beforeFallback` hook of the `Proxy` contract hasn't been removed.

To favor consistency, consider removing the `_beforeFallback` hook.

Update: *#4502* *5_0_133.*

> *_beforeFallback .*

CR-02 Wrong Visibility for a Public Constant

The `UPGRADE_INTERFACE_VERSION` variable has `internal` visibility. However, the docstrings clearly explain that this constant should be queried from the outside of the contract's context and so it should be public instead.

Consider changing the variable visibility to `public`.

Update: *#4382 121 5 .*

CR-03 Inconsistent `nonce` Enumeration in `AccessManager`

In the `AccessManager` contract, scheduled operations are assigned a `nonce` value which is sequentially incremented upon scheduling an `operationId`. This design allows distinguishing active and potentially cancelled operations that share the same `operationId` and is particularly useful in the `GovernorTimelockAccess` contract.

When successfully executing a scheduled operation, the whole `Schedule` struct of the corresponding `operationId` is being deleted, essentially resetting the `nonce`. Note that when canceling an operation the `nonce` entry of `Schedule` is not deleted. As a consequence, the invariant where the `nonce` is an incremental value for each `operationId` breaks.

This error could interfere with the proposals execution and cancellation in the `GovernorTimelockAccess` contract.

To fix this issue, consider retaining the `nonce` value when an operation is successfully executed through `AccessManager`.

Update: *#4603.*

CR-04 `AccessManager`'s `onlyAuthorized` Functions Cannot Be Executed Through `relay()`

The `AccessManager` contract provides two ways to execute a delay-restricted function after its scheduling period is completed:

1. directly call the restricted function

2. call the `relay` function so that the restricted function is called by the `AccessManager` contract itself

However, in the case of the `AccessManager`'s `onlyAuthorized` functions the execution through `relay` always fails. This is because the `onlyAuthorized` modifier only checks the permissions of `msg.sender` and nowhere considers the case where `address(this)` is the caller.

This issue is not severe within the context of `AccessManager` alone since there is always the option of a direct call to the `onlyAuthorized` function. However, this bug could result in failing governance proposals execution, because the `GovernorTimelockAccess` extension contract relies on `relay` for calling any function related to `AccessManager`.

Consider updating the `onlyAuthorized` modifier so that it also considers execution calls through the `relay` function.

Update: *#4612*.

Recommendations

[R01] Features and Design Suggestions

There are some features that are either missing or are partially implemented. We encourage the team to go over this list and evaluate, in each individual case, whether it is worth adding to the library:

- The `Nonces` contract misses a function that increments the nonce and returns the incremented one. Alternatively, the current function can be modified to return both the old and new nonce values. This should improve extensibility and give the implementers more choices on how they want to check nonce value correctness.
- The `MinimalForwarder` contract is not using `Nonces`. Consider using the already existing `Nonces` code in `MinimalForwarder`.
- Across the library, non-explicit imports are used. To improve readability and make the code easier to follow, especially when more definitions are inside the same file, consider using explicit imports instead.
- Across the library, mappings do not have named parameters for the keys and values. Consider adopting the newly available syntax to make the code easier to understand and follow.
- As it is right now, `ERC1155Supply` limits the `totalSupply` of all `tokenId`s cumulated all together with the `_totalSupplyAll` variable. Consider whether is worth exploring the possibility of having an `ERC1155SupplyAll` that reflects the current behavior and an `ERC1155Supply` that only caps individual `tokenId` amounts without cumulating them.

[R02] Overridable Functions Risk Classification

The codebase is a general-purpose library and for this, it has to be extensible and flexible enough to allow users to perform custom implementation on a variety of different use cases. For this reason, the vast majority of functions defined in all contracts are `virtual` so that those can be overridden.

However, in some special cases, overriding a function might be dangerous and undermine the inner mechanics of some established flows. The connection between overridable functions and

other parts of the codebase that make use of them assuming a specific behavior is not always easy to spot, and in general it doesn't mean that a user is not allowed to do a custom override, but that instead, it might unexpectedly introduce errors.

As such, we encourage the team to establish security levels for all overridable functions and to clearly classify those in the docstrings so that users are aware of the confidence that should be used when creating custom overridden implementations. We identified three main categories where all overridable functions can be grouped:

- Unrestricted overrideability: functions that can be overridden and are either unused by other parts of the codebase or if used, it's unlikely for a custom implementation, to break other mechanics that make use of them.
- Restricted overrideability: functions that can be overridden but special care should be taken if used in combination with other contracts or if some anti-patterns are already known so that users must avoid them.
- Discouraged overrideability: functions that can be overridden but that are likely to produce some issues when being re-implemented. Special care should be used by users when dealing with this category.

Some examples of restricted and discouraged levels functions are:

- The `nonces` getter of the `Nonces` library. We didn't identify any reason why someone would like to re-implement such a function. However, we noticed that if the getter returns something different from the actual nonce, the majority of contracts that make use of it will likely break or need a refactor to adopt such change.
- The `cap` function of the `ERC20Capped`, if re-implemented, will likely impact the `_update` function that makes use of it.
- The `_transferTokenUnits` function of `Votes` internally calls `_moveDelegateVotes` with `amount` assuming a 1:1 relation between token units and voting units, while the `_delegate` one calls the same but with `_getVotingUnits`. The `_getVotingUnits` is overridable and by default it just returns the `balanceOf(account)`, but if this were to be overridden to implement a quadratic or any constant product formula, the `_transferVotingUnits` will still mistakenly transfer `amount` because of the assumption. The `_getVotingUnits` should be of unrestricted category, but the actual codebase makes it possible to introduce severe issues if it was to be re-implemented.
- Sensitive `balanceOf` and `totalSupply` functions of `ERC20` can create issues on other contracts that make use of them and assume their default behavior.

When thinking about how to display categories to users, consider adopting docstrings of the following format

```
/// @Custom:overrideability { unrestricted | restricted | discouraged }
```

[R03] Testing and Fuzzing Opportunities - Phase 1

There are some functions in the library that might benefit from fuzzing. The following is a non-exhaustive list:

- The newly added `_isValidDescriptionForProposer` function.
- The entire `Checkpoints` and `DoubleEndedQueue` libraries.
- The majority of the functions in the `StorageSlot` and `Strings` libraries.

In other parts of the codebase, it would be beneficial to add specific test cases. Some examples are:

- Include a test to prove the correctness of this statement in the `TimelockController` contract.
- Similarly, consider conducting a test to prove that the chosen design in `ERC4626._deposit` is aligned with the statement in the docstrings.

[R04] Inextensible Choice of Admin Address

The latest design of the `TransparentUpgradeableProxy` uses an immutable `_admin` variable to avoid storage loads during the execution of the `_fallback` function, effectively skipping the usage of the `ERC1967` admin slot. This implies that there's no way to change the `_admin` that is allowed to upgrade the proxy.

Consider wrapping the `_admin` variable into a virtual function that can be overridden to use the `ERC1967` admin slot instead so that user can choose their own tradeoffs when it comes to using a transparent proxy.

[R05] Compatibility with EVM Chains Other Than Ethereum

While performing the audit we noticed that some of the patterns followed by the code base might not be compatible with chains in which Solidity code can be deployed.

Based on the official documentation of zkSync Era:

- `CREATE` and `CREATE2` on zkSync Era need to have the compiler to resolve the bytecode beforehand. Unfortunately, `Create2.deploy` takes it as input parameter and will not work if it's going to be implemented as an external user input. Similarly the `Clones` contract might be incompatible due to the same.
- There might be issues with `mload` and `mstore`. On zkSync Era, memory growth is in bytes, while on EVM is in words. The result of an `mstore(100,0)` will give an `msize` of `160` on EVM, of `132` on zkEVM. So any contract that doesn't deal with the `32 bytes = 1 word` growth will likely behave differently on EVM vs zkEVM. An example can be the `toETHSignedMessageHash` of `MessageHashUtils` or the `toString` of the `ShortStrings` contract.
- `CALLDATACOPY` and `CALLDATALOAD` will panic at `2^32-33` offset values. `_msgSender` from `ERC2771Context` might fail. The same is true for the `_delegate` function of `Proxy`.

Moreover, based on this other documentation page, another difference with zkSync Era is that `block.number` and `block.timestamp` within zkSync VM execution refers to the latest L1 batch that has been sent to Ethereum mainnet and not to the current L2 block number and timestamp. L1 batches might take some time to be finalized on L1 so the main effect is that `block.number` and timestamp would somehow be stuck to the same value until the next batch is finalized, definitely making time less continuous. Other rollups may have similar behaviors (Arbitrum). In terms of how this might affect the library let's imagine that an L1 batch has been pushed and another one will be added in a few minutes. In between the two batches:

- `Votes` will potentially break its mechanism. The `getPastVotes` function takes `timepoint` as reference. `timepoint` can be anything in the past but in this context it can't be something which is in the past and also after the latest L1 batch `timepoint`, otherwise the function will revert as if it was an attempt to read in the future.
- `TimelockController` operations might not move because the recorded timestamp refers to the previous L1 batch even if in the L2 VM context the timestamp is correct enough to move the proposals to another state.
- `VestingWallet` changes in vested amount are reflected on an L1-batches-basis instead of the L2 VM time base.

Notice that this is not a comprehensive list and more issues might emerge from such subtle differences.

Based on the official documentation of Optimism:

- `tx.origin` might be aliased (and so `msg.sender`) if it's an L1 -> L2 message. This might affect `Context`, `ERC2771Context` and `TransparentUpgradeableProxy`. This same issue applies to Arbitrum, and it's defined as address aliasing. The same happens on zkSync Era.

Based on the official documentation Polygon zkEVM:

- `block.number` returns the number of processable transactions, not an actual block number as in L2s. Any contract making use of it will have a wrong assumption of what the returned value is.

Notice that this is not a complete list of all the possible chains with their differences and is a mere list of examples that we were able to lift up by a rapid analysis. There might be other chains with completely different behaviours that we are not aware of.

It is important for the contracts team to establish a framework in which the library code can be tested on different VMs with their differences. Unfortunately, many alternative chains have different results if the information is queried through RPC calls or within VM execution directly (i.e., `block.number` or `chain.id`). For this, the suggestion is to test the contracts directly within VM executions using local setup nodes for each of the chains. There are no known tools that facilitate such operation, so we suggest further research on the topic.

[R06] Tokens Might Get Stuck in the Contract

The `transferFrom` and the `_transfer` functions of the `ERC721` contract allows the recipient to be `address(this)` since they don't perform the `onERC721Received` hook check. While the `ERC721Wrapper` provides a `_recover` method for such scenario, the `ERC721` would end up with tokens effectively stuck in the contract.

Consider whether is worth adding an internal `_recover` function or prohibiting `address(this)` as a recipient.

In general, the issue has been officially raised in Phase 1 too but we decided to add recommendations instead because we think it's a general issue that should be solved by taking a common decision and we defer to the team on how to proceed from here.

[R07] Testing and Fuzzing Opportunities - Phase 4

There are some functions in the library that might benefit from fuzzing. The following is a non-exhaustive list:

- The entire `SignedMath` library.

[R08] Overlapping Operation of Multiple Delay Mechanisms in `AccessManager`

There are currently three different delay mechanisms in the `AccessManager` contract:

- Execution delays: access to restricted functions is allowed to members of an authorized group with respect to each member's `executionDelay`. This means that in order to execute a restricted function for which they are authorized, a member should first schedule the operation and then trigger its execution after `executionDelay` period of time has passed. This delay is set per member.
- Group granting delays: the delay between the time point when an account is granted membership and the time point that the member becomes active, in essence allowed to perform or schedule a restricted operation. This delay is set per group.
- Class admin delays: these delays are also considered for some sensitive operations at the `AccessManager` contract that fall under no delay restriction other than the caller's own `executionDelay`.

The operation field of these three mechanisms could overlap in some cases so that it is hard to extract strong constraints about the delay restrictions applied to a set of sensitive operations, e.g. calling a restricted function on the target contract, granting membership to authorized groups, or configuring the `AccessManager` contract.

For example, an authorized member can access some restricted function `functionA` with `executionDelayA`, but could possibly access `functionA` faster if they are also admins of `functionA`'s groups and can grant membership to a new account with zero `executionDelay` for this new member. The admin will be able to access `functionA` faster if their `executionDelay` for granting membership and the `grantDelay` for this group are overall lower than `executionDelayA`.

The administrators of the `AccessManager` contract should take these edge cases into consideration when configuring the delay values. However, it's admittedly not trivial to configure all its parameters in a secure and effective manner.

The design could be made clearer and simpler if grant delays are omitted in favor of minimum execution delays per group that apply to all groups' members. However, there are some cases where zero execution delay is useful. For example, administrative entities that take decisions by running a voting procedure, like DAOs, would be expected to perform a single step in order to execute restricted functionality. Thus, this design simplification should be accompanied by a solution for cases like a DAO administrator. For example, scheduled operations that have passed the delay period could be open for anyone to execute or the `schedule` function could accept an additional `address` parameter for the scheduler to provide a trusted account to `relay` the scheduled operation on their behalf.

Consider reviewing the above design suggestions and try to simplify the delay mechanisms so that it becomes more comprehensive, needing less administrative effort to manage effectively and securely and allowing special entities (like DAOs) to interact efficiently.

Conclusion

Version 5 is a new milestone not only for the development team but also for OpenZeppelin as a whole. The level of appreciation that previous versions of the library have shown is outstanding and the number of protocols and projects that rely on it is always increasing. Because of this, there has been an enormous effort from many team members to make this possible and deliver impactful changes that we hope can allow for more robust and new projects to be built.

As the security services team we are happy to be part of it and we particularly value the research, the study and the continuous back and forth with changes and new feature that the development team have been through, but also the engagement in the endless discussions and brainstorming sessions with us.

Waiting to see what the world will build with it, we wish all the best to this new release!