Scan Findings:

Report generated by auditbase.com

Issues

Centralization risk for trusted owners

Description:

Having a single EOA as the only owner of contracts is a large centralization risk and a single point of failure. A single private key may be taken in a hack, or the sole holder of the key may become unable to retrieve the key when necessary. Consider changing to a multi-signature setup, or having a role-based authorization model.

Severity:

M

Snippet:

```
File: contracts/TokenStaking.sol

59          function updateDistributeRate(uint256 newMultiplier) external
onlyOwner accrueReward {
64          function setStartTimeForStaking(uint256 startAt) external
onlyOwner {
```

Contracts are vulnerable to fee-on-transfer-token-related accounting issues

Description:

Without measuring the balance before and after the transfer, there's no way to ensure that enough tokens were transferred, in the cases where the token has a fee-on-transfer mechanic. If there are latent funds in the contract, subsequent transfers will succeed.

Severity:

M

Snippet:

```
File: contracts/TokenPresale.sol

115         function buyTokens(address payToken, uint256 amountIn) external
nonReentrant saleActive {
116             require(paymentTokens[payToken].isAllowed, "Unsupported
payment token");
117             require(amountIn > 0, "Invalid amount");
118
119             uint256 tokenAmount = calculateTokenAmount(payToken,
amountIn);
120
121             // Transfer stable token to contract
122             IERC20(payToken).transferFrom(msg.sender, address(this),
amountIn);
123
124             // Transfer sale tokens to buyer
125             saleToken.transfer(msg.sender, tokenAmount);
126             tokensSold += tokenAmount;
127
128             emit Purchased(msg.sender, payToken, amountIn, tokenAmount);
129         }
```

ERC20: unsafe use of `transfer()` / `transferFrom()`

Description:

Use OpenZeppelin’s `SafeERC20`'s `safeTransfer()` / `safeTransferFrom()` instead `transfer()` / `transferFrom()`. Some tokens do not implement the ERC20 standard properly but are still accepted by most code that accepts ERC20 tokens. For

example Tether (USDT)'s `transfer()` and `transferFrom()` functions on L1 do not return booleans as the specification requires, and instead have no return value. When these sorts of tokens are cast to `IERC20`, their function signatures do not match and therefore the calls made, revert (see this link for a test case).

Severity:

M

Snippet:

```
File: contracts/TokenStaking.sol

80              stakingToken.transferFrom(msg.sender, address(this),
amount);
110             stakingToken.transfer(msg.sender, info.amount);
127             stakingToken.transfer(msg.sender, pending);
```

Missing checks for the L2 Sequencer grace period

Description:

Chainlink recommends that users using price oracles, check whether the grace period has passed in the event the Arbitrum Sequencer goes down. To help your applications identify when the sequencer is unavailable, you can use a data feed that tracks the last known status of the sequencer at a given point in time. This helps you prevent mass liquidations by providing a grace period to allow customers to react to such an event.

Severity:

M

Snippet:

```
File: contracts/TokenPresale.sol

74          function getTokenPriceInUSD(address token) public view returns
(uint256) {
75              require(paymentTokens[token].isAllowed, "Token not
allowed");
76              if (paymentTokens[token].isStable) {
77                  return 1e8;
```

```
78              }
79
80              (, int256 price,,,) =
AggregatorV3Interface(paymentTokens[token].priceFeed).latestRoundData();
81              require(price > 0, "Invalid price");
82
83              return uint256(price); // 8 decimals typically
84          }
```

Insufficient oracle validation

Description:

There is no freshness check on the timestamp of the prices fetched from the Chainlink oracle, so old prices may be used if OCR was unable to push an update in time. Add a staleness threshold number of seconds configuration parameter, and ensure that the price fetched is from within that time range.

Severity:

M

Snippet:

```
File: contracts/TokenPresale.sol

80              (, int256 price,,,) =
AggregatorV3Interface(paymentTokens[token].priceFeed).latestRoundData();
```

Unsafe use of `transfer()`/`transferFrom()` with `IERC20`

Description:

Some tokens do not implement the ERC20 standard properly but are still accepted by most code that accepts ERC20 tokens. For example Tether (USDT)'s `transfer()` and `transferFrom()` functions on L1 do not return booleans as the specification requires, and instead have no return value. When these sorts of tokens are cast to `IERC20`, their function signatures do not match and therefore the calls made, revert (see this link for a test case). Use OpenZeppelinundefineds `SafeERC20`'s `safeTransfer()`/`safeTransferFrom()` instead.

Severity:

M

Snippet:

```
File: contracts/TokenPresale.sol

146             IERC20(token).transfer(to, bal);
122             IERC20(payToken).transferFrom(msg.sender, address(this),
amountIn);
```

Some ERC20 revert on zero value transfer

Description:

Example: https://github.com/d-xo/weird-erc20#revert-on-zero-value-transfers.

Severity:

M

Snippet:

```
File: contracts/TokenStaking.sol

80              stakingToken.transferFrom(msg.sender, address(this),
amount);
110             stakingToken.transfer(msg.sender, info.amount);
127             stakingToken.transfer(msg.sender, pending);
```

Chainlink oracle will return the wrong price if the aggregator hits minAnswer

Description:

Chainlink aggregators have a built-in circuit breaker if the price of an asset goes outside of a predetermined price band.The result is that if an asset experiences a huge drop in value (i.e. LUNA crash) the price of the oracle will continueto return the minPrice instead of the actual price of the asset. This would allow users to continue borrowing with theasset but at the wrong price. This is exactly

what happened to Venus on BSC when LUNA crashed.Consider adding Min/Max price check.

Severity:

M

Snippet:

```
File: contracts/TokenPresale.sol

74          function getTokenPriceInUSD(address token) public view returns
(uint256) {
75              require(paymentTokens[token].isAllowed, "Token not
allowed");
76              if (paymentTokens[token].isStable) {
77                  return 1e8;
78              }
79
80              (, int256 price,,,) =
AggregatorV3Interface(paymentTokens[token].priceFeed).latestRoundData();
81              require(price > 0, "Invalid price");
82
83              return uint256(price); // 8 decimals typically
84          }
```

Large transfers may not work with some ERC20 tokens

Description:

Some IERC20 implementations (e.g UNI, COMP) may fail if the valued transferred is larger than uint96. Source

Severity:

M

Snippet:

```
File: contracts/TokenStaking.sol

80              stakingToken.transferFrom(msg.sender, address(this),
amount);
110             stakingToken.transfer(msg.sender, info.amount);
```

```
127             stakingToken.transfer(msg.sender, pending);
```

Return values of `transfer` / `transferFrom` not checked.

Description:

Not all `IERC20` implementations `revert` when there's a failure in `transfer` / `transferFrom`. The function signature has a `boolean` return value and they indicate errors that way instead. By not checking the return value, operations that should have marked as failed, may potentially go through without actually making a payment

Severity:

M

Snippet:

```
File: contracts/TokenStaking.sol

80              stakingToken.transferFrom(msg.sender, address(this),
amount);
110             stakingToken.transfer(msg.sender, info.amount);
127             stakingToken.transfer(msg.sender, pending);
```

Using isContract is not a reliable method to determine whether an address is an EOA (Externally Owned Account) or a contract.

Description:

Using isContract to differentiate EOAs from contracts in Ethereum is unreliable due to proxy contracts and contract creation during transactions. OpenZeppelin's Address library deprecated this method, acknowledging its limitations. For accuracy, alternative checks should be used to avoid false classifications.

Severity:

M

Snippet:

```
File: @openzeppelin/contracts-upgradeable/proxy/utils/Initializable.sol

87                  (isTopLevelCall && _initialized < 1) || (!
AddressUpgradeable.isContract(address(this)) && _initialized == 1),
```

`abi.encodePacked()` should not be used with dynamic types when passing the result to a hash function such as `keccak256()`

Description:

Use `abi.encode()` instead which will pad items to 32 bytes, which will prevent hash collisions (e.g. `abi.encodePacked(0x123,0x456)` => `0x123456` => `abi.encodePacked(0x1,0x23456)`, but `abi.encode(0x123,0x456)` => `0x0...1230...456`). Unless there is a compelling reason, `abi.encode` should be preferred. If there is only one argument to `abi.encodePacked()` it can often be cast to `bytes()` or `bytes32()` instead. If all arguments are strings and or bytes, `bytes.concat()` should be used instead.

Severity:

L

Snippet:

```
File: contracts/TokenAirdrop.sol

bytes32 leaf = keccak256(abi.encodePacked(msg.sender, amount));
```

Initializers could be front-run

Description:

Initializers could be front-run, allowing an attacker to either set their own values, take ownership of the contract, and in the best case forcing a re-deployment

Severity:

L

Snippet:

```
File: contracts/MetaTxGateway.sol

87          function initialize() public initializer {
88              __Ownable_init();
89              __ReentrancyGuard_init();
90              __Pausable_init();
91              __UUPSUpgradeable_init();
92          }
```

Loss of precision due to division by large numbers

Description:

Division by large numbers may result in the result being zero, due to solidity not supporting fractions. Consider requiring a minimum amount for the numerator to ensure that it is always larger than the denominator.

Severity:

L

Snippet:

```
File: contracts/TokenStaking.sol

188             return getDistributionRate() * 365 * 1e18 /
totalStakes.stakingShare;
```

Array lengths not checked

Description:

If the length of the arrays are not required to be of the same length, user operations may not be fully executed due to a mismatch in the number of items iterated over, versus the number of items provided in the second array.

Severity:

L

Snippet:

```
File: @openzeppelin/contracts/utils/cryptography/MerkleProof.sol

178         function multiProofVerify(
179             bytes32[] memory proof,
180             bool[] memory proofFlags,
181             bytes32 root,
182             bytes32[] memory leaves
183         ) internal pure returns (bool) {
184             return processMultiProof(proof, proofFlags, leaves) == root;
185         }
264         function multiProofVerify(
265             bytes32[] memory proof,
266             bool[] memory proofFlags,
267             bytes32 root,
268             bytes32[] memory leaves,
269             function(bytes32, bytes32) view returns (bytes32) hasher
270         ) internal view returns (bool) {
271             return processMultiProof(proof, proofFlags, leaves, hasher)
== root;
272         }
352         function multiProofVerifyCalldata(
353             bytes32[] calldata proof,
354             bool[] calldata proofFlags,
355             bytes32 root,
356             bytes32[] memory leaves
357         ) internal pure returns (bool) {
358             return processMultiProofCalldata(proof, proofFlags, leaves)
```

```
== root;
359         }
438         function multiProofVerifyCalldata(
439             bytes32[] calldata proof,
440             bool[] calldata proofFlags,
441             bytes32 root,
442             bytes32[] memory leaves,
443             function(bytes32, bytes32) view returns (bytes32) hasher
444         ) internal view returns (bool) {
445             return processMultiProofCalldata(proof, proofFlags, leaves,
hasher) == root;
446         }
```

Division by zero not prevented

Description:

The divisions below take an input parameter which does not have any zero-value checks, which may lead to the functions reverting when zero is passed.

Severity:

L

Snippet:

```
File: contracts/GasCreditVault.sol

391                 uint256 tokenAmount = (deltaCredits *
contractBalance) / tokenValueInCredits;
```

Missing checks for `address(0x0)` when assigning values to address state variables

Description:

This issue arises when an address state variable is assigned a value without a preceding check to ensure it isn't address(0x0). This can lead to unexpected behavior as address(0x0) often represents an uninitialized address.

Severity:

L

Snippet:

```
File: @openzeppelin/contracts/access/Ownable.sol

97              _owner = newOwner;
```

Upgradeable contract is missing a __gap[50] storage variable to allow for new storage variables in later versions

Description:

This issue arises when an upgradeable contract doesn't include a __gap[50] storage variable. This variable is crucial for facilitating future upgrades and preserving storage layout.

Severity:

L

Snippet:

```
File: contracts/MetaTxGateway.sol

19      contract MetaTxGateway is Initializable, OwnableUpgradeable,
ReentrancyGuardUpgradeable, PausableUpgradeable, UUPSUpgradeable {
```

Use `Ownable2Step` rather than `Ownable`

Description:

`Ownable2Step` and `Ownable2StepUpgradeable` prevent the contract ownership from mistakenly being transferred to an address that cannot handle it (e.g. due to a typo in the address), by requiring that the recipient of the owner permissions actively accept via a contract call of its own.

Severity:

L

Snippet:

```
File: contracts/TokenStaking.sol

7       contract TokenStaking is Ownable {
```

Unsafe downcast

Description:

When a type is downcast to a smaller type, the higher order bits are truncated, effectively applying a modulo to the original value. Without any other checks, this wrapping will lead to unexpected behavior and bugs.

Severity:

L

Snippet:

```
File: @openzeppelin/contracts/utils/Strings.sol

413                 return (s, address(uint160(v)));
```

Upgradeable contract not initialized

Description:

Upgradeable contracts are initialized via an initializer function rather than by a constructor. Leaving such a contract uninitialized may lead to it being taken over by a malicious user.

Severity:

L

Snippet:

```
File: contracts/MetaTxGateway.sol
```

```
19      contract MetaTxGateway is Initializable, OwnableUpgradeable,
ReentrancyGuardUpgradeable, PausableUpgradeable, UUPSUpgradeable {
```

Arrays can grow in size without a way to shrink them

Description:

Array entries are added but are never removed. Consider whether this should be the case, or whether there should be a maximum, or whether old entries should be removed. Consider adding a `pop` function to remove entries.

Severity:

L

Snippet:

```
File: contracts/TokenStaking.sol

82              userStakes[msg.sender].push(StakeInfo({
```

File allows a version of solidity that is susceptible to an assembly optimizer bug

Description:

In solidity versions 0.8.13 and 0.8.14, there is an optimizer bug where, if the use of a variable is in a separate assembly block from the block in which it was stored, the mstore operation is optimized out, leading to uninitialized memory. The code currently does not have such a pattern of execution, but it does use mstores in assembly blocks, so it is a risk for future changes. The affected solidity versions should be avoided if at all possible.

Severity:

L

Snippet:

```
File: @openzeppelin/contracts/utils/Strings.sol

470             assembly ("memory-safe") {
471                 mstore(output, outputLength)
472                 mstore(0x40, add(output, shl(5, shr(5,
add(outputLength, 63)))))
473             }
```

Functions calling contracts/addresses with transfer hooks are missing reentrancy guards

Description:

Even if the function follows the best practice of check-effects-interaction, not using a reentrancy guard when there may be transfer hooks to unknown or untrusted ERC20 tokens will open the users of this protocol up to read-only reentrancies with no way to protect against it, except by block-listing the whole protocol.

Severity:

L

Snippet:

```
File: contracts/TokenPresale.sol

146             IERC20(token).transfer(to, bal);
```

Signature Malleability of EVM's ecrecover

Description:

The function calls the Solidity ecrecover() function directly to verify the given signatures. However, the ecrecover() EVM opcode allows malleable (non-unique) signatures and thus is susceptible to replay attacks. Consider using the OpenZeppelin ECDSA library instead, which provides a safe ecrecover() wrapper that guarantees unique signatures.

Severity:

L

Snippet:

```
File: @openzeppelin/contracts/utils/cryptography/ECDSA.sol

148             address signer = ecrecover(hash, v, r, s);
```

Constant decimal values

Description:

The use of fixed decimal values such as 1e18 or 1e8 in Solidity contracts can lead to inaccuracies, bugs, and vulnerabilities, particularly when interacting with tokens having different decimal configurations. Not all ERC20 tokens follow the standard 18 decimal places, and assumptions about decimal places can lead to miscalculations. Always retrieve and use the decimals() function from the token contract itself when performing calculations involving token amounts.

Severity:

L

Snippet:

```
File: contracts/TokenStaking.sol

125             info.rewardDebt = (info.stakingShare *
totalStakes.rewardIndex) / 1e18;
135             uint256 accumulated = (info.stakingShare *
totalStakes.rewardIndex) / 1e18;
145             userStake.rewardDebt = (share * totalStakes.rewardIndex) /
1e18;
163                 totalStakes.rewardIndex += (reward * 1e18) /
totalStakes.stakingShare;
176             return rewardMultiplier * DISTRIBITE_UNIT/ 1e18;
188             return getDistributionRate() * 365 * 1e18 /
totalStakes.stakingShare;
```

No limits when setting state variable amounts

Description:

It is important to ensure state variables numbers are set to a reasonable value.

Severity:

L

Snippet:

```
File: contracts/TokenStaking.sol

60              rewardMultiplier = newMultiplier;
```

Pausing withdrawals is unfair to the users

Description:

Users should always have the possibility of accessing their own funds, but when these functions are paused, their funds will be locked until the contracts are unpaused.

Severity:

L

Snippet:

```
File: contracts/GasCreditVault.sol

276         function withdraw(address token, uint256 creditAmount) external
whenNotPaused onlyStablecoin(token) {
```

Owner can renounce ownership while system is paused

Description:

The contract owner is not prevented from renouncing the ownership while the contract is paused, which would cause any user assets stored in the protocol to be locked indefinitely.

Severity:

L

Snippet:

```
File: contracts/GasCreditVault.sol

197         function pause() external onlyOwner {
```

Double type casts create complexity within the code

Description:

Double type casting should be avoided in Solidity contracts to prevent unintended consequences and ensure accurate data representation. Performing multiple type casts in succession can lead to unexpected truncation, rounding errors, or loss of precision, potentially compromising the contract's functionality and reliability. Furthermore, double type casting can make the code less readable and harder to maintain, increasing the likelihood of errors and misunderstandings during development and debugging. To ensure precise and consistent data handling, developers should use appropriate data types and avoid unnecessary or excessive type casting, promoting a more robust and dependable contract execution.

Severity:

L

Snippet:

```
File: @openzeppelin/contracts/utils/Strings.sol

return toHexString(uint256(uint160(addr)), ADDRESS_LENGTH);
return (s, address(uint160(v)));
```

For loops in `public` or `external` functions should be avoided due to high gas costs and possible DOS

Description:

In Solidity, for loops can potentially cause Denial of Service (DoS) attacks if not handled carefully. DoS attacks can occur when an attacker intentionally exploits the gas cost of a function, causing it to run out of gas or making it too expensive for other users to call. Below are some scenarios where for loops can lead to DoS attacks: Nested for loops can become exceptionally gas expensive and should be used sparingly.

Severity:

L

Snippet:

```
File: contracts/TeamAllocation.sol

31          function allocateTokens(
32              address[] calldata _wallets,
33              string[] calldata names,
34              uint256[] calldata amounts
35          ) external onlyOwner {
36              require(
37                  _wallets.length == names.length && names.length ==
amounts.length,
38                  "Mismatched input lengths"
39              );
40
41              for (uint256 i = 0; i < _wallets.length; ++i) {
42                  wallets.push(_wallets[i]);
43                  membersInfo.push(Member({
44                      name: names[i],
45                      balance: amounts[i],
46                      startTime: block.timestamp,
47                      withdrawn: 0
48                  }));
```

```
49                  totalAllocated += amounts[i];
50              }
51          }
```

Function calls within for loops

Description:

Making function calls within loops in Solidity can lead to inefficient gas usage, potential bottlenecks, and increased vulnerability to attacks. Each function call or external call consumes gas, and when executed within a loop, the gas cost multiplies, potentially causing the transaction to run out of gas or exceed block gas limits. This can result in transaction failure or unpredictable behavior.

Severity:

L

Snippet:

```
File: contracts/TokenStaking.sol

216             for (uint256 i = 0; i < stakes.length; ++i) {
217                 stakes[i].pendingReward =
calculateClaimableReward(stakes[i]);
218             }
```

External calls in an unbounded for-loop may result in a DoS

Description:

Consider limiting the number of iterations in for-loops that make external calls.

Severity:

L

Snippet:

```
File: contracts/TeamAllocation.sol

41              for (uint256 i = 0; i < _wallets.length; ++i) {
```

Contracts are not using their OZ Upgradeable counterparts

Description:

The non-upgradeable standard version of OpenZeppelin's library is inherited/used by the contracts. It would be safer to use the upgradeable versions of the library contracts to avoid unexpected behavior.

Use the contracts from `@openzeppelin/contracts-upgradeable` instead of `@openzeppelin/contracts` where applicable. See https://github.com/OpenZeppelin/openzeppelin-contracts-upgradeable/tree/master/contracts for a list of available upgradeable contracts

Severity:

L

Snippet:

```
File: contracts/TokenStaking.sol

4       import "node_modules/@openzeppelin/contracts/token/ERC20/
IERC20.sol";
5       import "./node_modules/@openzeppelin/contracts/access/Ownable.sol";
```

Multiplication on the result of a division

Description:

Dividing an integer by another integer will often result in loss of precision. When the result is multiplied by another number, the loss of precision is magnified, often to material magnitudes. (X / Z) * Y should be re-written as (X * Y) / Z.

Severity:

L

Snippet:

```
File: @openzeppelin/contracts/utils/math/Math.sol

193                 return SafeCast.toUint(a > 0) * ((a - 1) / b + 1);
```

Consider implementing two-step procedure for updating protocol addresses

Description:

Lack of two-step procedure for critical operations leaves them error-prone. Consider adding two step procedure on the critical functions. See similar findings in previous Code4rena contests for reference: https://code4rena.com/reports/2022-06-illuminate/#2-critical-changes-should-use-two-step-procedure

Severity:

L

Snippet:

```
File: contracts/MetaTxGateway.sol

101         function setRelayerAuthorization(address relayer, bool
authorized) external onlyOwner {
```

`decimals()` is not a part of the ERC-20 standard

Description:

The `decimals()` function is not a part of the ERC-20 standard, and was added later as an optional extension. As such, some valid ERC20 tokens do not support this interface, so it is unsafe to blindly cast all tokens to this interface, and then call this function.

Severity:

L

Snippet:

```
File: contracts/GasCreditVault.sol

418             uint8 tokenDecimals = IERC20Metadata(token).decimals();
448             uint8 tokenDecimals = IERC20Metadata(token).decimals();
```

Governance functions should be controlled by time locks

Description:

Governance functions (such as upgrading contracts, setting critical parameters) should be controlled using time locks to introduce a delay between a proposal and its execution. This gives users time to exit before a potentially dangerous or malicious operation is applied.

Severity:

L

Snippet:

```
File: contracts/TokenStaking.sol

64          function setStartTimeForStaking(uint256 startAt) external
onlyOwner {
59          function updateDistributeRate(uint256 newMultiplier) external
onlyOwner accrueReward {
```

Missing checks for address(0x0) when updating address state variables

Description:

issing checks for address(0x0) when updating address state variables

Severity:

L

Snippet:

```
File: @openzeppelin/contracts/utils/cryptography/EIP712.sol

76              _cachedThis = address(this);
```

The additions/multiplications may silently overflow because they're in unchecked blocks with no preceding value checks, which may lead to unexpected results.

Description:

The additions/multiplications may silently overflow because they're in unchecked blocks with no preceding value checks, which may lead to unexpected results.

Severity:

L

Snippet:

```
File: contracts/TokenPresale.sol

100                     amount / (10 ** (from - to)) :
101                     amount * (10 ** (to - from));
```

Unbounded state array which is iterated upon

Description:

Iterating over an unbounded state array in Solidity can result in excessive gas consumption, especially if the array size exceeds the block gas limit. This issue commonly arises in tasks like token distribution. To address this, it is recommended to limit array sizes for iteration, consider alternative data

structures like linked lists, adopt paginated processing for smaller batches over multiple transactions, or use a 'state array' with a separate index-tracking array to manage large datasets and avoid gas-related problems.

Severity:

L

Snippet:

```
File: contracts/TeamAllocation.sol

96                  if (wallets[i] == wallet) return i;
```

Subtraction may underflow if multiplication is too large

Description:

Severity:

L

Snippet:

```
File: @openzeppelin/contracts/utils/math/Math.sol

261                 inverse *= 2 - denominator * inverse; // inverse mod 2⁸
262                 inverse *= 2 - denominator * inverse; // inverse mod 2¹⁶
263                 inverse *= 2 - denominator * inverse; // inverse mod 2³²
264                 inverse *= 2 - denominator * inverse; // inverse mod 2⁶⁴
265                 inverse *= 2 - denominator * inverse; // inverse mod
2¹²⁸
266                 inverse *= 2 - denominator * inverse; // inverse mod
2²⁵⁶
345                         gcd - remainder * quotient
354                         x - y * int256(quotient)
```

Consider disallowing minting/transfers to address(this)

Description:

A tranfer to the token contract itself is unlikely to be correct and more likely to be a common user error due to a copy & paste mistake. Proceeding with such a transfer will result in the permanent loss of user tokens.

Severity:

L

Snippet:

```
File: contracts/mock/MockERC20.sol

14          function mint(address to, uint256 amount) external {
15              _mint(to, amount);
16          }
```

Constants should be defined rather than using magic numbers

Description:

Even assembly can benefit from using readable constants instead of hex/numeric literals.

Severity:

NC

Snippet:

```
File: contracts/TokenStaking.sol

46          uint256 public rewardMultiplier = 1e18;
125             info.rewardDebt = (info.stakingShare *
totalStakes.rewardIndex) / 1e18;
135             uint256 accumulated = (info.stakingShare *
totalStakes.rewardIndex) / 1e18;
145             userStake.rewardDebt = (share * totalStakes.rewardIndex) /
```

```
1e18;
163                 totalStakes.rewardIndex += (reward * 1e18) /
totalStakes.stakingShare;
176             return rewardMultiplier * DISTRIBITE_UNIT/ 1e18;
181             return amount * getLockMultiplier(duration) *
getAmountMultiplier(amount) / 1e4;
188             return getDistributionRate() * 365 * 1e18 /
totalStakes.stakingShare;
192             if (duration == LockDuration.QUATERLY) return YEAR / 4;
199             if (duration == LockDuration.QUATERLY) return 125;
200             if (duration == LockDuration.HALF_YEARLY) return 150;
201             if (duration == LockDuration.YEARLY) return 200;
202             return 100;
206             if (amount >= 500_000 ether) return 250;
207             if (amount >= 200_000 ether) return 200;
208             if (amount >= 100_000 ether) return 150;
209             if (amount >= 50_000 ether) return 125;
210             if (amount >= 20_000 ether) return 115;
211             return 100;
```

Use scientific notation (e.g. 1e18) rather than exponentiation (e.g. 10**18)

Description:

While the compiler knows to optimize away the exponentiation, it's still better coding practice to use idioms that do not require compiler optimization, if they exist.

Severity:

NC

Snippet:

```
File: contracts/TokenPresale.sol

100                     amount / (10 ** (from - to)) :
101                     amount * (10 ** (to - from));
```

Event is not properly indexed

Description:

Index event fields make the field more quickly accessible to off-chain tools that parse events. This is especially useful when it comes to filtering based on an address. However, note that each index field costs extra gas during emission, so it's not necessarily best to index the maximum allowed per event (three fields). Where applicable, each event should use three indexed fields if there are three or more fields, and gas usage is not particularly of concern for the events in question. If there are fewer than three applicable fields, all of the applicable fields should be indexed.

Severity:

NC

Snippet:

```
File: @openzeppelin/contracts-upgradeable/security/PausableUpgradeable.sol

22          event Paused(address account);
27          event Unpaused(address account);
```

Function ordering does not follow the Solidity style guide

Description:

According to the Solidity style guide, functions should be laid out in the following order : `constructor()` , `receive()` , `fallback()` , `external` , `public` , `internal` , `private` , but the cases below do not follow this pattern.

Severity:

NC

Snippet:

```
File: contracts/TokenStaking.sol

/// @auditbase internal functions should not come before external functions
```

```
/// @auditbase internal functions should not come before external functions
    function updateReward() external accrueReward {
```

Imports could be organized more systematically

Description:

This issue arises when the contract's interface is not imported first, followed by each of the interfaces it uses, followed by all other files.

Severity:

NC

Snippet:

```
File: contracts/TokenPresale.sol

5       import "node_modules/@openzeppelin/contracts/token/ERC20/
IERC20.sol";
```

Constants in comparisons should appear on the left side

Description:

This issue arises when constants in comparisons appear on the right side, which can lead to typo bugs.

Severity:

NC

Snippet:

```
File: contracts/TokenStaking.sol

65              require(stakingStartedAt == 0, 'Staking already started');
72              require(stakingStartedAt != 0 && block.timestamp >=
stakingStartedAt, 'Staking not started');
78              require(amount > 0, "Amount must be > 0");
121             require(pending > 0, "No reward available");
153             if (totalStakes.lastUpdatedAt == 0)
```

```
156             if (block.timestamp > totalStakes.lastUpdatedAt &&
totalStakes.stakingShare > 0) {
185             if (totalStakes.stakingShare == 0)
206             if (amount >= 500_000 ether) return 250;
207             if (amount >= 200_000 ether) return 200;
208             if (amount >= 100_000 ether) return 150;
209             if (amount >= 50_000 ether) return 125;
210             if (amount >= 20_000 ether) return 115;
```

else-block not required

Description:

One level of nesting can be removed by not having an else block when the if-block returns

Severity:

NC

Snippet:

```
File: @openzeppelin/contracts/utils/Strings.sol

279             if (absSuccess && absValue < ABS_MIN_INT256) {
280                 return (true, negativeSign ? -int256(absValue) :
int256(absValue));
281             } else if (absSuccess && negativeSign && absValue ==
ABS_MIN_INT256) {
282                 return (true, type(int256).min);
283             } else return (false, 0);
281             } else if (absSuccess && negativeSign && absValue ==
ABS_MIN_INT256) {
282                 return (true, type(int256).min);
283             } else return (false, 0);
410             if (end - begin == expectedLength) {
411                 // length guarantees that this does not overflow, and
value is at most type(uint160).max
412                 (bool s, uint256 v) =
_tryParseHexUintUncheckedBounds(input, begin, end);
413                 return (s, address(uint160(v)));
414             } else {
415                 return (false, address(0));
416             }
```

Events may be emitted out of order due to reentrancy

Description:

Ensure that events follow the best practice of check-effects-interaction, and are emitted before external calls

Severity:

NC

Snippet:

```
File: contracts/TokenStaking.sol

/// @audit transfer() called before event
112             emit Withdrawn(msg.sender, stakeIndex, info.amount);

/// @audit transferFrom() called before event
96              emit Staked(msg.sender, amount, duration);

/// @audit transfer() called before event
128             emit ClaimedReward(msg.sender, stakeIndex, pending);
```

If-statement can be converted to a ternary

Description:

The code can be made more compact while also increasing readability by converting the following if-statements to ternaries (e.g. foo += (x > y) ? a : b)

Severity:

NC

Snippet:

```
File: @openzeppelin/contracts/utils/Strings.sol

281             } else if (absSuccess && negativeSign && absValue ==
ABS_MIN_INT256) {
282                 return (true, type(int256).min);
```

```
283             } else return (false, 0);
```

Import declarations should import specific identifiers, rather than the whole file

Description:

Using import declarations of the form import {} from 'some/file.sol' avoids polluting the symbol namespace making flattened files smaller, and speeds up compilation

Severity:

NC

Snippet:

```
File: contracts/TokenStaking.sol

4       import "node_modules/@openzeppelin/contracts/token/ERC20/
IERC20.sol";
5       import "./node_modules/@openzeppelin/contracts/access/Ownable.sol";
```

Adding a return statement when the function defines a named return variable, is redundant

Description:

If a function defines a named return variable, it is not necessary to explicitly return it. It will automatically be returned at the end of the function.

Severity:

NC

Snippet:

```
File: contracts/MetaTxGateway.sol

156             return success;
166             return success;
```

```
178             return totalValue;
241             return successes;
```

Public functions not called by the contract should be declared external instead

Description:

Contracts are allowed to override their parents' functions and change the visibility from `external` to `public`.

Severity:

NC

Snippet:

```
File: contracts/TokenStaking.sol

184         function calculateAPR() public view returns(uint256) {
185             if (totalStakes.stakingShare == 0)
186                 return 0;
187 
188             return getDistributionRate() * 365 * 1e18 /
totalStakes.stakingShare;
189         }
```

Constant redefined elsewhere

Description:

Consider defining in only one contract so that values cannot become out of sync when only one location is updated. A cheap way to store constants in a single location is to create an internal constant in a library. If the variable is a local cache of another contract's value, consider making the cache variable internal or private, which will require external users to query the contract with the source of truth, so that callers don't get out of sync.

Severity:

NC

Snippet:

```
File: contracts/TokenAirdrop.sol

    IERC20 public immutable token;
```

Multiple `address`/ID mappings can be combined into a single `mapping` of an `address`/ID to a `struct`, for readability

Description:

Well-organized data structures make code reviews easier, which may lead to fewer bugs. Consider combining related mappings into mappings to structs, so it's clear what data is related

Severity:

NC

Snippet:

```
File: contracts/GasCreditVault.sol

55          mapping(address => TokenInfo) public tokenInfo;
56          mapping(address => uint256) public credits;
57          mapping(address => mapping(address => uint256)) public
creditsInToken;
```

Duplicated `require()`/`revert()` checks should be refactored to a modifier or function

Description:

The compiler will inline the function, which will avoid JUMP instructions usually associated with functions.

Severity:

NC

Snippet:

```
File: contracts/TeamAllocation.sol

55              require(index != NOT_FOUND, "No Member wallet");
```

Interfaces should be indicated with an I prefix in the contract name

Description:

Severity:

NC

Snippet:

```
File: contracts/TokenPresale.sol

8       interface AggregatorV3Interface {
```

Variables need not be initialized to zero

Description:

The default value for variables is zero, so initializing them to zero is superfluous.

Severity:

NC

Snippet:

```
File: contracts/TokenStaking.sol

216             for (uint256 i = 0; i < stakes.length; ++i) {
```

Variable names for `constant`s are improperly named

Description:

According to the Style guide, for `constant` variable names, each word should use all capital letters, with underscores separating each word (CONSTANT_CASE).

Severity:

NC

Snippet:

```
File: contracts/TokenStaking.sol

41          uint256 constant totalRewardTokens = 100_000_000 ether;
```

Variable names for `immutable`s should use CONSTANT_CASE

Description:

For `immutable` variable names, each word should use all capital letters, with underscores separating each word (CONSTANT_CASE).

Severity:

NC

Snippet:

```
File: contracts/TokenStaking.sol

8           IERC20 public immutable stakingToken;
```

Lines are too long

Description:

Usually lines in source code are limited to 80 characters. Today's screens are much larger so it's reasonable to stretch this in some cases. The solidity style guide recommends a maximumum line length of 120 characters, so the lines below should be split when they reach that length.

Severity:

NC

Snippet:

```
File: contracts/TokenStaking.sol

require(block.timestamp >= info.startTime +
getLockDuration(info.lockDuration) || stakingEnded, "Lock not expired");
```

File is missing NatSpec comments

Description:

The file does not contain any of the NatSpec comments (@inheritdoc, @param, @return, @notice), which are important for documentation and user confirmation.

Severity:

NC

Snippet:

```
File: contracts/TokenStaking.sol

// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

import "node_modules/@openzeppelin/contracts/token/ERC20/IERC20.sol";
import "./node_modules/@openzeppelin/contracts/access/Ownable.sol";

contract TokenStaking is Ownable {
    IERC20 public immutable stakingToken;
```

```
    struct StakeInfo {
        uint256 amount;
        uint256 startTime;
        uint256 endTime;
        LockDuration lockDuration;
        uint256 stakingShare;
        uint256 rewardDebt;
        uint256 claimedReward;
        uint256 pendingReward;
        uint256 totalAccumulatedReward;
        bool withdrawn;
    }

    struct TotalStakeInfo {
        uint256 totalStaked;
        uint256 stakingShare;
        uint256 rewardIndex;
        uint256 lastUpdatedAt;
        uint256 totalRewardDistributed;
        uint256 lastDistributedAt;
    }

    enum LockDuration {
        NORMAL,
        QUATERLY,
        HALF_YEARLY,
        YEARLY
    }

    uint256 constant YEAR = 365 days;
    uint256 constant DISTRIBITE_UNIT = 10_000 ether;
    uint256 constant totalRewardTokens = 100_000_000 ether;

    mapping(address => StakeInfo[]) public userStakes;
    TotalStakeInfo public totalStakes;

    uint256 public rewardMultiplier = 1e18;
    uint256 public stakingStartedAt;
    bool public stakingEnded;

    event RewardMultiplierUpdated(uint256);
    event Staked(address indexed user, uint256 stakedAmount, LockDuration
duration);
    event Withdrawn(address indexed user, uint256 stakeIndex, uint256
amount);
    event ClaimedReward(address indexed user, uint256 stakeIndex, uint256
reward);

    constructor(address _stakingToken, address _owner) Ownable(_owner) {
        stakingToken = IERC20(_stakingToken);
    }

    function updateDistributeRate(uint256 newMultiplier) external onlyOwner
accrueReward {
        rewardMultiplier = newMultiplier;
        emit RewardMultiplierUpdated(newMultiplier);
```

```
    }

    function setStartTimeForStaking(uint256 startAt) external onlyOwner {
        require(stakingStartedAt == 0, 'Staking already started');
        require(startAt > block.timestamp, 'Invalid timestamp');

        stakingStartedAt = startAt;
    }

    modifier stakingActive() {
        require(stakingStartedAt != 0 && block.timestamp >=
stakingStartedAt, 'Staking not started');
        require(!stakingEnded, 'Staking ended');
        _;
    }

    function stake(uint256 amount, LockDuration duration) external
stakingActive {
        require(amount > 0, "Amount must be > 0");

        stakingToken.transferFrom(msg.sender, address(this), amount);

        userStakes[msg.sender].push(StakeInfo({
            amount: amount,
            startTime: block.timestamp,
            endTime: block.timestamp + getLockDuration(duration),
            lockDuration: duration,
            stakingShare: 0,
            rewardDebt: 0,
            claimedReward: 0,
            pendingReward: 0,
            totalAccumulatedReward: 0,
            withdrawn: false
        }));

        updateStakingShare(msg.sender);
        emit Staked(msg.sender, amount, duration);
    }

    function withdraw(uint256 stakeIndex) external {
        StakeInfo storage info = userStakes[msg.sender][stakeIndex];
        require(!info.withdrawn, "Already withdrawn");
        require(block.timestamp >= info.startTime +
getLockDuration(info.lockDuration) || stakingEnded, "Lock not expired");

        claimReward(stakeIndex);
        // Update total stakes (subtract the withdrawn amount and share)
        info.withdrawn = true;
        totalStakes.totalStaked -= info.amount;
        totalStakes.stakingShare -= info.stakingShare;

        stakingToken.transfer(msg.sender, info.amount);

        emit Withdrawn(msg.sender, stakeIndex, info.amount);
    }
```

```
    function claimReward(uint256 stakeIndex) public accrueReward {
        require(stakeIndex < userStakes[msg.sender].length, "Invalid stake
index");
        StakeInfo storage info = userStakes[msg.sender][stakeIndex];
        require(!info.withdrawn, "Stake already withdrawn");

        uint256 pending = calculateClaimableReward(info);
        require(pending > 0, "No reward available");

        // Update all reward tracking fields
        info.claimedReward += pending;
        info.rewardDebt = (info.stakingShare * totalStakes.rewardIndex) /
1e18;

        stakingToken.transfer(msg.sender, pending);
        emit ClaimedReward(msg.sender, stakeIndex, pending);
    }

    function calculateClaimableReward(StakeInfo memory info) public view
returns (uint256) {
        if (info.withdrawn)
            return 0;

        uint256 accumulated = (info.stakingShare *
totalStakes.rewardIndex) / 1e18;
        if (accumulated < info.rewardDebt) return 0;
        return accumulated - info.rewardDebt;
    }

    function updateStakingShare(address user) internal accrueReward {
        StakeInfo storage userStake = userStakes[user]
[userStakes[user].length - 1];
        uint256 share = calculateStakingShare(userStake.amount,
userStake.lockDuration);

        // Set reward debt before increasing total share
        userStake.rewardDebt = (share * totalStakes.rewardIndex) / 1e18;

        totalStakes.totalStaked += userStake.amount;
        totalStakes.stakingShare += share;
        userStake.stakingShare = share;
    }

    modifier accrueReward() {
        if (totalStakes.lastUpdatedAt == 0)
            totalStakes.lastUpdatedAt = block.timestamp;

        if (block.timestamp > totalStakes.lastUpdatedAt &&
totalStakes.stakingShare > 0) {
            uint256 duration = block.timestamp - totalStakes.lastUpdatedAt;
            uint256 reward = (duration * getDistributionRate()) / 1 days;
            if (reward + totalStakes.totalRewardDistributed >
totalRewardTokens) {
                reward = totalRewardTokens -
totalStakes.totalRewardDistributed;
                stakingEnded = true;
```

```
            }
            totalStakes.rewardIndex += (reward * 1e18) /
totalStakes.stakingShare;
            totalStakes.lastUpdatedAt = block.timestamp;
            totalStakes.totalRewardDistributed += reward;
        }
        _;
    }

    function updateReward() external accrueReward {

    }

    // this is reward amount per day
    function getDistributionRate() internal view returns(uint256) {
        return rewardMultiplier * DISTRIBITE_UNIT/ 1e18;
    }


    function calculateStakingShare(uint256 amount, LockDuration duration)
internal pure returns(uint256) {
        return amount * getLockMultiplier(duration) *
getAmountMultiplier(amount) / 1e4;
    }

    function calculateAPR() public view returns(uint256) {
        if (totalStakes.stakingShare == 0)
            return 0;

        return getDistributionRate() * 365 * 1e18 /
totalStakes.stakingShare;
    }

    function getLockDuration(LockDuration duration) public pure returns
(uint256) {
        if (duration == LockDuration.QUATERLY) return YEAR / 4;
        if (duration == LockDuration.HALF_YEARLY) return YEAR / 2;
        if (duration == LockDuration.YEARLY) return YEAR;
        return 0;
    }

    function getLockMultiplier(LockDuration duration) public pure returns
(uint256) {
        if (duration == LockDuration.QUATERLY) return 125;
        if (duration == LockDuration.HALF_YEARLY) return 150;
        if (duration == LockDuration.YEARLY) return 200;
        return 100;
    }

    function getAmountMultiplier(uint256 amount) public pure returns
(uint256) {
        if (amount >= 500_000 ether) return 250;
        if (amount >= 200_000 ether) return 200;
        if (amount >= 100_000 ether) return 150;
        if (amount >= 50_000 ether) return 125;
        if (amount >= 20_000 ether) return 115;
```

```
        return 100;
    }

    function getStakes(address user) external view returns (StakeInfo[]
memory stakes) {
        stakes = userStakes[user];
        for (uint256 i = 0; i < stakes.length; ++i) {
            stakes[i].pendingReward = calculateClaimableReward(stakes[i]);
        }
    }

    function getTotalStakes() external view returns (TotalStakeInfo memory)
{
        return totalStakes;
    }
}
```

Function declarations should have NatSpec descriptions

Description:

Function declarations should be preceded by a NatSpec comment.

Severity:

NC

Snippet:

```
File: contracts/TokenStaking.sol

55          constructor(address _stakingToken, address _owner)
Ownable(_owner) {
59          function updateDistributeRate(uint256 newMultiplier) external
onlyOwner accrueReward {
64          function setStartTimeForStaking(uint256 startAt) external
onlyOwner {
77          function stake(uint256 amount, LockDuration duration) external
stakingActive {
99          function withdraw(uint256 stakeIndex) external {
115         function claimReward(uint256 stakeIndex) public accrueReward {
131         function calculateClaimableReward(StakeInfo memory info) public
view returns (uint256) {
140         function updateStakingShare(address user) internal accrueReward
{
170         function updateReward() external accrueReward {
175         function getDistributionRate() internal view returns(uint256) {
180         function calculateStakingShare(uint256 amount, LockDuration
```

```
duration) internal pure returns(uint256) {
184         function calculateAPR() public view returns(uint256) {
191         function getLockDuration(LockDuration duration) public pure
returns (uint256) {
198         function getLockMultiplier(LockDuration duration) public pure
returns (uint256) {
205         function getAmountMultiplier(uint256 amount) public pure
returns (uint256) {
214         function getStakes(address user) external view returns
(StakeInfo[] memory stakes) {
221         function getTotalStakes() external view returns (TotalStakeInfo
memory) {
```

Contract declarations should have `@notice` tags

Description:

`@notice` is used to explain to end users what the contract does, and the compiler interprets `///` or `/**` comments as this tag if one wasn't explicitly provided.

Severity:

NC

Snippet:

```
File: contracts/TokenStaking.sol

7       contract TokenStaking is Ownable {
```

Error declarations should have NatSpec descriptions

Description:

Severity:

NC

Snippet:

```
File: @openzeppelin/contracts/utils/Strings.sol

30          error StringsInsufficientHexLength(uint256 value, uint256
length);
35          error StringsInvalidChar();
40          error StringsInvalidAddressFormat();
```

Use @inheritdoc rather than using a non-standard annotation

Description:

Instead of referring to other code with @dev See {some-code}, use @inheritdoc.

Severity:

NC

Snippet:

```
File: @openzeppelin/contracts/token/ERC20/ERC20.sol

82           * @dev See {IERC20-totalSupply}.
89           * @dev See {IERC20-balanceOf}.
96           * @dev See {IERC20-transfer}.
110          * @dev See {IERC20-allowance}.
117          * @dev See {IERC20-approve}.
133          * @dev See {IERC20-transferFrom}.
```

Contract does not follow the Solidity style guide's suggested layout ordering

Description:

The style guide says that, within a contract, the ordering should be 1) Type declarations, 2) State variables, 3) Events, 4) Modifiers, and 5) Functions, but the contract(s) below do not follow this ordering.

Severity:

NC

Snippet:

```
File: contracts/TokenStaking.sol

71          modifier stakingActive() {
152         modifier accrueReward() {
```

Non-external/public variable names should begin with an underscore

Description:

According to the Solidity Style Guide, non-external/public variable names should begin with an underscore

Severity:

NC

Snippet:

```
File: contracts/GasCreditVault.sol

52          EnumerableSet.AddressSet private whitelistedTokens;
53          EnumerableSet.AddressSet private relayers;
```

Consider disabling `renounceOwnership()`

Description:

If the plan for your project does not include eventually giving up all ownership control, consider overwriting OpenZeppelin's Ownable's `renounceOwnership()` function in order to disable it.

Severity:

NC

Snippet:

```
File: contracts/TokenStaking.sol

7       contract TokenStaking is Ownable {
```

Not using the named return variables anywhere in the function is confusing

Description:

Consider changing the return variable to be an unnamed one, since the variable is never assigned, nor is it returned by name

Severity:

NC

Snippet:

```
File: contracts/MetaTxGateway.sol

261         ) internal view returns (bool valid) {
300         function _buildDomainSeparator() internal view returns (bytes32
domainSeparator) {
317         function calculateRequiredValue(MetaTransaction[] calldata
metaTxs) external pure returns (uint256 totalValue) {
326         function getNonce(address user) external view returns (uint256
currentNonce) {
335         function isRelayerAuthorized(address relayer) external view
returns (bool isAuthorized) {
343         function getDomainSeparator() external view returns (bytes32
separator) {
351         function getMetaTransactionTypehash() external pure returns
(bytes32 typehash) {
359         function getMainTypehash() external pure returns (bytes32
typehash) {
376         ) external view returns (bytes32 digest) {
411         function getTotalBatchCount() external view returns (uint256
count) {
419         function getVersion() external pure returns (string memory
version) {
```

Unused arguments in override functions

Description:

Arguments in overridden functions that are unused should have their names removed or commented out to avoid compiler warnings.

Severity:

NC

Snippet:

```
File: contracts/MetaTxGateway.sol

430         function _authorizeUpgrade(address newImplementation) internal
override onlyOwner {}
```

Non-library/interface files should use fixed compiler versions, not floating ones

Description:

Using a floating compiler version like ^0.8.16 or >=0.8.16 can lead to unexpected behavior if the compiler version used differs from the one intended. It's recommended to specify a fixed compiler version for non-library/interface files.

Severity:

NC

Snippet:

```
File: contracts/TokenStaking.sol

2       pragma solidity ^0.8.20;
```

Expressions for constant values such as a call to `keccak256()`, should use `immutable` rather than `constant`

Description:

While it **doesn't save any gas** because the compiler knows that developers often make this mistake, it's still best to use the right tool for the task at hand. There is a difference between `constant` variables and `immutable` variables, and they should each be used in their appropriate contexts. `constants` should be used for literal values written into the code, and `immutable` variables should be used for expressions, or values calculated in, or passed into the constructor.

Severity:

NC

Snippet:

```
File: contracts/MetaTxGateway.sol

23          bytes32 private constant EIP712_DOMAIN_TYPEHASH = keccak256(
24              "EIP712Domain(string name,string version,uint256
chainId,address verifyingContract)"
25          );
28          bytes32 private constant META_TRANSACTION_STRUCT_TYPEHASH =
keccak256(
29              "MetaTransaction(address to,uint256 value,bytes data)"
30          );
33          bytes32 private constant META_TRANSACTION_TYPEHASH = keccak256(
34              "MetaTransactions(address from,MetaTransaction[]
metaTxs,uint256 nonce,uint256 deadline)MetaTransaction(address to,uint256
value,bytes data)"
35          );
```

Empty Function Body - Consider commenting why

Description:

Severity:

NC

Snippet:

```
File: contracts/TokenStaking.sol

170         function updateReward() external accrueReward {
171
172         }
```

Contracts should have full test coverage

Description:

While 100% code coverage does not guarantee that there are no bugs, it often will catch easy-to-find bugs, and will ensure that there are fewer regressions when the code invariably has to be modified. Furthermore, in order to get full coverage, code authors will often have to re-organize their code so that it is more modular, so that each component can be tested separately, which reduces interdependencies between modules and layers, and makes for code that is easier to reason about and audit.

Severity:

NC

Snippet:

```
File: Various Files

None
```

Large or complicated code bases should implement invariant tests

Description:

Large code bases, or code with lots of inline-assembly, complicated math, or complicated interactions between multiple contracts, should implement invariant fuzzing tests. Invariant fuzzers such as Echidna require the test writer to come up with invariants which should not be violated under any circumstances, and the fuzzer tests various inputs and function calls to ensure that the invariants always hold. Even code with 100% code coverage can still

have bugs due to the order of the operations a user performs, and invariant fuzzers, with properly and extensively-written invariants, can close this testing gap significantly.

Severity:

NC

Snippet:

```
File: Various Files

None
```

Long functions should be refactored into multiple, smaller, functions

Description:

Severity:

NC

Snippet:

```
File: contracts/MetaTxGateway.sol

190         function executeMetaTransactions(
191             address from,
192             MetaTransaction[] calldata metaTxs,
193             bytes calldata signature,
194             uint256 nonce,
195             uint256 deadline
196         ) external payable nonReentrant whenNotPaused returns (bool[]
memory successes) {
197             require(authorizedRelayers[msg.sender], "Unauthorized
relayer");
198             require(block.timestamp <= deadline, "Transaction expired");
199             require(nonce == nonces[from], "Invalid nonce");
200             require(_verifySignature(from, metaTxs, signature, nonce,
deadline), "Invalid signature");
201
202             require(metaTxs.length > 0, "Empty batch Txs");
203
204             // Calculate total value required for all meta-transactions
205             uint256 totalValueRequired = 0;
206             if (msg.value > 0) {
```

```
207                 totalValueRequired = _calculateTotalValue(metaTxs);
208                 require(msg.value == totalValueRequired, "Incorrect
native token amount");
209             }
210 
211             successes = new bool[](metaTxs.length);
212 
213             // Store batch transaction log
214             uint256 batchId = nextBatchId++;
215             uint256 valueUsed = 0;
216 
217             // Execute all transactions in the batch
218             for (uint256 i = 0; i < metaTxs.length; i++) {
219                 bool success = _executeMetaTransaction(from,
metaTxs[i]);
220 
221                 // Track value used for each transaction
222                 if (success) {
223                     valueUsed += metaTxs[i].value;
224                 }
225             }
226 
227             // Refund unused native tokens if any transactions failed
228             if (totalValueRequired > 0) {
229                 uint256 refundAmount = totalValueRequired - valueUsed;
230                 if (refundAmount > 0) {
231                     (bool refundSuccess, ) = payable(from).call{value:
refundAmount}("");
232                     require(refundSuccess, "Refund failed");
233                 }
234 
235                 emit NativeTokenUsed(batchId, totalValueRequired,
valueUsed, refundAmount);
236             }
237 
238             // Increment nonce to prevent replay
239             nonces[from]++;
240 
241             return successes;
242         }
```

Strings should use double quotes rather than single quotes

Description:

See the Solidity Style Guide

Severity:

NC

Snippet:

```
File: contracts/TokenStaking.sol

72              require(stakingStartedAt != 0 && block.timestamp >=
stakingStartedAt, 'Staking not started');
65              require(stakingStartedAt == 0, 'Staking already started');
73              require(!stakingEnded, 'Staking ended');
66              require(startAt > block.timestamp, 'Invalid timestamp');
```

`address` shouldn't be hard-coded

Description:

It is often better to declare addresses as `immutable`, and assign them via constructor arguments. This allows the code to remain the same across deployments on different networks and avoids recompilation when addresses need to change.

Severity:

NC

Snippet:

```
File: contracts/mock/MockUSDT.sol

9               _mint(0xa9315C1C008c022c4145E993eC9d1a3AF73D0A62, 1000_000
ether);
```

Consider using `block.number` instead of `block.timestamp`

Description:

`block.timestamp` is vulnerable to miner manipulation and creates a potential front-running vulnerability. Consider using `block.number` instead.

Severity:

NC

Snippet:

```
File: contracts/TokenStaking.sol

66              require(startAt > block.timestamp, 'Invalid timestamp');
72              require(stakingStartedAt != 0 && block.timestamp >=
stakingStartedAt, 'Staking not started');
84                  startTime: block.timestamp,
85                  endTime: block.timestamp + getLockDuration(duration),
102             require(block.timestamp >= info.startTime +
getLockDuration(info.lockDuration) || stakingEnded, "Lock not expired");
154                 totalStakes.lastUpdatedAt = block.timestamp;
156             if (block.timestamp > totalStakes.lastUpdatedAt &&
totalStakes.stakingShare > 0) {
157                 uint256 duration = block.timestamp -
totalStakes.lastUpdatedAt;
164                 totalStakes.lastUpdatedAt = block.timestamp;
```

Consider bounding input array length

Description:

The functions below take in an unbounded array, and make function calls for entries in the array. While the function will revert if it eventually runs out of gas, it may be a nicer user experience to `require()` that the length of the array is below some reasonable maximum, so that the user doesn't have to use up a full transaction's gas only to see that the transaction reverts.

Severity:

NC

Snippet:

```
File: contracts/MetaTxGateway.sol

266             for (uint256 i = 0; i < metaTxs.length; i++) {
267                 metaTxHashes[i] = keccak256(abi.encode(
268                     META_TRANSACTION_STRUCT_TYPEHASH,
269                     metaTxs[i].to,
270                     metaTxs[i].value,
271                     keccak256(metaTxs[i].data)
```

```
272                 ));
273             }
381             for (uint256 i = 0; i < metaTxs.length; i++) {
382                 metaTxHashes[i] = keccak256(abi.encode(
383                     META_TRANSACTION_STRUCT_TYPEHASH,
384                     metaTxs[i].to,
385                     metaTxs[i].value,
386                     keccak256(metaTxs[i].data)
387                 ));
388             }
```

Large numeric literals should use underscores for readability

Description:

At a glance, it's quite difficult to understand how big this number is. Use underscores to make values more clear.

Severity:

NC

Snippet:

```
File: contracts/GasCreditVault.sol

64          uint256 public constant PRICE_FEED_TIMEOUT = 3600; // 1 hour
for stale price protection
```

Implement some type of version counter that will be incremented automatically for contract upgrades

Description:

As part of the upgradeability of Proxies , the contract can be upgraded multiple times, where it is a systematic approach to record the version of each upgrade. For instance, you could implement this: uint256 public authorizeUpgradeCounter; function _authorizeUpgrade(address newImplementation) internal { authorizeUpgradeCounter+=1; }

Severity:

NC

Snippet:

```
File: contracts/MetaTxGateway.sol

430         function _authorizeUpgrade(address newImplementation) internal
override onlyOwner {}
```

Cast to `bytes` or `bytes32` for clearer semantic meaning

Description:

Using a cast on a single argument, rather than `abi.encodePacked()` makes the intended operation more clear, leading to less reviewer confusion.

Severity:

NC

Snippet:

```
File: contracts/MetaTxGateway.sol

279                 keccak256(abi.encodePacked(metaTxHashes)),
394                 keccak256(abi.encodePacked(metaTxHashes)),
```

Variables should be named in mixedCase style

Description:

As the Solidity Style Guide suggests: arguments, local variables and mutable state variables should be named in mixedCase style.

Severity:

NC

Snippet:

```
File: contracts/TokenPresale.sol

108             uint256 tokenUSDPrice = getTokenPriceInUSD(payToken);
```

Consider using named mappings

Description:

Consider moving to solidity version 0.8.18 or later, and using named mappings to make it easier to understand the purpose of each mapping.

Severity:

NC

Snippet:

```
File: contracts/TokenPresale.sol

32          mapping(address => PaymentToken) public paymentTokens;
```

Use allowlist/denylist rather than whitelist/blacklist

Description:

Use alternative variants, e.g. allowlist/denylist instead of whitelist/blacklist.

Severity:

NC

Snippet:

```
File: contracts/GasCreditVault.sol

38          event TokenWhitelisted(address indexed token, address
priceFeed);
52          EnumerableSet.AddressSet private whitelistedTokens;
```

```
87           * @dev Modifier to restrict function access to whitelisted
relayers only
90               require(relayers.contains(msg.sender), "Caller not
whitelisted relayers");
121          * @dev Adds a new relayer to the whitelist
124          * @notice Relayer address must not be zero and not already
whitelisted
126          function addWhitelistedRelayer(address relayer) external
onlyOwner {
128              require(!relayers.contains(relayer), "Relayer already
whitelisted");
136          * @dev Removes a relayer from the whitelist
139          * @notice Relayer must be currently whitelisted
141          function removeWhitelistedRelayer(address relayer) external
onlyOwner {
142              require(relayers.contains(relayer), "Relayer not
whitelisted");
150          * @dev Adds a token to the whitelist with price feed
configuration
155          * @notice Token must not already be whitelisted
158          function whitelistToken(
164              require(!whitelistedTokens.contains(token), "Token already
whitelisted");
167              whitelistedTokens.add(token);
173              emit TokenWhitelisted(token, priceFeed);
177          * @dev Removes a token from the whitelist
181          * @notice Token must be currently whitelisted
185              require(whitelistedTokens.contains(token), "Token not
whitelisted");
186              whitelistedTokens.remove(token);
218          * @notice Withdraws all balances of whitelisted tokens
221              address[] memory tokens = whitelistedTokens.values();
249          * @notice Token must be whitelisted and amount must be greater
than zero
254              require(whitelistedTokens.contains(token), "Token not
whitelisted");
297          * @notice Only whitelisted relayers can call this function
311              // Iterate over whitelisted tokens and deduct credited and
deposited accordingly
312              address[] memory tokens = whitelistedTokens.values();
346              address[] memory tokens = whitelistedTokens.values();
377              address[] memory tokens = whitelistedTokens.values();
497          * @dev Returns the list of all whitelisted token addresses
498          * @return Array of whitelisted token addresses
500          function getWhitelistedTokens() external view returns
(address[] memory) {
501              return whitelistedTokens.values();
505          * @dev Returns the list of all whitelisted relayer addresses
506          * @return Array of whitelisted relayer addresses
508          function getWhitelistedRelayers() external view returns
(address[] memory) {
533          * @dev Checks if a token is whitelisted
535          * @return Whether the token is whitelisted
537          function isTokenWhitelisted(address token) external view
returns (bool) {
538              return whitelistedTokens.contains(token);
```

```
542          * @dev Checks if an address is a whitelisted relayer
544          * @return Whether the address is a whitelisted relayer
546         function isRelayerWhitelisted(address relayer) external view
returns (bool) {
```

Function names should use lowerCamelCase

Description:

According to the Solidity style guide function names should be in `mixedCase` (lowerCamelCase).

Severity:

NC

Snippet:

```
File: contracts/TokenStaking.sol

184         function calculateAPR() public view returns(uint256) {
```

Consider adding a deny-list

Description:

Doing so will significantly increase centralization, but will help to prevent hackers from using stolen tokens.

Severity:

NC

Snippet:

```
File: contracts/TokenStaking.sol

7       contract TokenStaking is Ownable {
```

Custom errors should be used rather than `revert()` / `require()`

Description:

Custom errors are available from solidity version 0.8.4. Custom errors are more easily processed in `try` - `catch` blocks, and are easier to re-use and maintain.

Severity:

NC

Snippet:

```
File: contracts/TokenStaking.sol

65              require(stakingStartedAt == 0, 'Staking already started');
66              require(startAt > block.timestamp, 'Invalid timestamp');
72              require(stakingStartedAt != 0 && block.timestamp >=
stakingStartedAt, 'Staking not started');
73              require(!stakingEnded, 'Staking ended');
78              require(amount > 0, "Amount must be > 0");
101             require(!info.withdrawn, "Already withdrawn");
102             require(block.timestamp >= info.startTime +
getLockDuration(info.lockDuration) || stakingEnded, "Lock not expired");
116             require(stakeIndex < userStakes[msg.sender].length,
"Invalid stake index");
118             require(!info.withdrawn, "Stake already withdrawn");
121             require(pending > 0, "No reward available");
```

Top level declarations should be separated by two blank lines

Description:

Severity:

NC

Snippet:

```
File: contracts/TokenPresale.sol
```

```
14
```

Interfaces should be defined in separate files from their usage

Description:

This issue arises when the interfaces are defined in the same files where they are used. They should be separated into different files for better readability and reusability.

Severity:

NC

Snippet:

```
File: contracts/TokenPresale.sol

8       interface AggregatorV3Interface {
9           function latestRoundData()
10              external
11              view
12              returns (uint80, int256 answer, uint256, uint256, uint80);
13      }
```

Custom error has no error details

Description:

Consider adding parameters to the error to indicate which user or values caused the failure.

Severity:

NC

Snippet:

```
File: @openzeppelin/contracts/utils/Strings.sol
```

```
35          error StringsInvalidChar();
40          error StringsInvalidAddressFormat();
```

Overridden function has no body

Description:

Consider adding a NatSpec comment describing why the function doesn't need a body and or the purpose it serves.

Severity:

NC

Snippet:

```
File: contracts/MetaTxGateway.sol

430         function _authorizeUpgrade(address newImplementation) internal
override onlyOwner {}
```

Consider moving `msg.sender` checks to a common authorization `modifier`

Description:

Severity:

NC

Snippet:

```
File: contracts/TokenStaking.sol

116             require(stakeIndex < userStakes[msg.sender].length,
"Invalid stake index");
```

Memory-safe annotation preferred over comment variant

Description:

The memory-safe annotation (`assembly ("memory-safe") { ... }`), available starting in Solidity version 0.8.13 is preferred over the comment variant, which will be removed in a future breaking release. The comment variant is only meant for externalized library code that needs to work in earlier versions (e.g. `SafeTransferLib` needs to be able to be used in many different versions).

Severity:

NC

Snippet:

File: @openzeppelin/contracts-upgradeable/utils/StorageSlotUpgradeable.sol

```
133             /// @solidity memory-safe-assembly
134             assembly {
135                 r.slot := store.slot
136             }
113             /// @solidity memory-safe-assembly
114             assembly {
115                 r.slot := store.slot
116             }
123             /// @solidity memory-safe-assembly
124             assembly {
125                 r.slot := slot
126             }
63              /// @solidity memory-safe-assembly
64              assembly {
65                  r.slot := slot
66              }
103             /// @solidity memory-safe-assembly
104             assembly {
105                 r.slot := slot
106             }
83              /// @solidity memory-safe-assembly
84              assembly {
85                  r.slot := slot
86              }
93              /// @solidity memory-safe-assembly
94              assembly {
95                  r.slot := slot
96              }
73              /// @solidity memory-safe-assembly
74              assembly {
75                  r.slot := slot
```

```
76             }
```

Unused `error` definition

Description:

Note that there may be cases where an error superficially appears to be used, but this is only because there are multiple definitions of the error in different files. In such cases, the error definition should be moved into a separate file. The instances below are the unused definitions.

Severity:

NC

Snippet:

```
File: @openzeppelin/contracts/interfaces/draft-IERC6093.sol

    error ERC721InvalidOwner(address owner);
    error ERC721NonexistentToken(uint256 tokenId);
    error ERC721IncorrectOwner(address sender, uint256 tokenId, address
owner);
    error ERC721InvalidSender(address sender);
    error ERC721InvalidReceiver(address receiver);
    error ERC721InsufficientApproval(address operator, uint256 tokenId);
    error ERC721InvalidApprover(address approver);
    error ERC721InvalidOperator(address operator);
    error ERC1155InsufficientBalance(address sender, uint256 balance,
uint256 needed, uint256 tokenId);
    error ERC1155InvalidSender(address sender);
    error ERC1155InvalidReceiver(address receiver);
    error ERC1155MissingApprovalForAll(address operator, address owner);
    error ERC1155InvalidApprover(address approver);
    error ERC1155InvalidOperator(address operator);
    error ERC1155InvalidArrayLength(uint256 idsLength, uint256
valuesLength);
```

Events are missing sender information

Description:

When an action is triggered based on a user's action, not being able to filter based on who triggered the action makes event processing a lot more cumbersome. Including the msg.sender the events of these types of action will

make events much more useful to end users. Include `msg.sender` in the event output.

Severity:

NC

Snippet:

```
File: contracts/TokenStaking.sol

61              emit RewardMultiplierUpdated(newMultiplier);
```

Imports should use double quotes rather than single quotes

Description:

According to the documentation imports should use a double quote instead of a single one.

Severity:

NC

Snippet:

```
File: contracts/mock/MockUSDT.sol

4       import './MockERC20.sol';
```

Enum values should be used in place of constant array indexes

Description:

Create a commented enum value to use in place of constant array indexes, this makes the code far easier to understand.

Severity:

NC

Snippet:

```
File: @openzeppelin/contracts/utils/Strings.sol

87              buffer[0] = "0";
88              buffer[1] = "x";
```

Zero as a function argument should have a descriptive meaning

Description:

Consider using descriptive constants or an enum instead of passing zero directly on function calls, as that might be error-prone, to fully describe the caller's intention.

Severity:

NC

Snippet:

```
File: contracts/TokenStaking.sol

82              userStakes[msg.sender].push(StakeInfo({
83                  amount: amount,
84                  startTime: block.timestamp,
85                  endTime: block.timestamp + getLockDuration(duration),
86                  lockDuration: duration,
87                  stakingShare: 0,
88                  rewardDebt: 0,
89                  claimedReward: 0,
90                  pendingReward: 0,
91                  totalAccumulatedReward: 0,
92                  withdrawn: false
93              }));
```

Function names should differ to make the code more readable

Description:

In Solidity, while function overriding allows for functions with the same name to coexist, it is advisable to avoid this practice to enhance code readability and maintainability. Having multiple functions with the same name, even with different parameters or in inherited contracts, can cause confusion and increase the likelihood of errors during development, testing, and debugging. Using distinct and descriptive function names not only clarifies the purpose and behavior of each function, but also helps prevent unintended function calls or incorrect overriding. By adopting a clear and consistent naming convention, developers can create more comprehensible and maintainable smart contracts.

Severity:

NC

Snippet:

```
File: contracts/TokenStaking.sol

    function withdraw(uint256 stakeIndex) external {
        StakeInfo storage info = userStakes[msg.sender][stakeIndex];
        require(!info.withdrawn, "Already withdrawn");
        require(block.timestamp >= info.startTime +
getLockDuration(info.lockDuration) || stakingEnded, "Lock not expired");

        claimReward(stakeIndex);
        // Update total stakes (subtract the withdrawn amount and share)
        info.withdrawn = true;
        totalStakes.totalStaked -= info.amount;
        totalStakes.stakingShare -= info.stakingShare;

        stakingToken.transfer(msg.sender, info.amount);

        emit Withdrawn(msg.sender, stakeIndex, info.amount);
    }
```

Use EIP-5627 to describe EIP-712 domains

Description:

EIP-5267 is a standard which allows for the retrieval and description of EIP-712 hash domains. This enable external tools to allow users to view the fields and values that describe their domain.

Severity:

NC

Snippet:

```
File: contracts/MetaTxGateway.sol

23          bytes32 private constant EIP712_DOMAIN_TYPEHASH = keccak256(
24              "EIP712Domain(string name,string version,uint256
chainId,address verifyingContract)"
25          );
```

It is standard for all external and public functions to be override from an interface

Description:

This is to ensure the whole API is extracted in an interface

Severity:

NC

Snippet:

```
File: contracts/TokenStaking.sol

59          function updateDistributeRate(uint256 newMultiplier) external
onlyOwner accrueReward {
60              rewardMultiplier = newMultiplier;
61              emit RewardMultiplierUpdated(newMultiplier);
62          }
64          function setStartTimeForStaking(uint256 startAt) external
onlyOwner {
65              require(stakingStartedAt == 0, 'Staking already started');
```

```
66              require(startAt > block.timestamp, 'Invalid timestamp');
67
68              stakingStartedAt = startAt;
69          }
77          function stake(uint256 amount, LockDuration duration) external
stakingActive {
78              require(amount > 0, "Amount must be > 0");
79
80              stakingToken.transferFrom(msg.sender, address(this),
amount);
81
82              userStakes[msg.sender].push(StakeInfo({
83                  amount: amount,
84                  startTime: block.timestamp,
85                  endTime: block.timestamp + getLockDuration(duration),
86                  lockDuration: duration,
87                  stakingShare: 0,
88                  rewardDebt: 0,
89                  claimedReward: 0,
90                  pendingReward: 0,
91                  totalAccumulatedReward: 0,
92                  withdrawn: false
93              }));
94
95              updateStakingShare(msg.sender);
96              emit Staked(msg.sender, amount, duration);
97          }
99          function withdraw(uint256 stakeIndex) external {
100             StakeInfo storage info = userStakes[msg.sender][stakeIndex];
101             require(!info.withdrawn, "Already withdrawn");
102             require(block.timestamp >= info.startTime +
getLockDuration(info.lockDuration) || stakingEnded, "Lock not expired");
103
104             claimReward(stakeIndex);
105             // Update total stakes (subtract the withdrawn amount and
share)
106             info.withdrawn = true;
107             totalStakes.totalStaked -= info.amount;
108             totalStakes.stakingShare -= info.stakingShare;
109
110             stakingToken.transfer(msg.sender, info.amount);
111
112             emit Withdrawn(msg.sender, stakeIndex, info.amount);
113         }
115         function claimReward(uint256 stakeIndex) public accrueReward {
116             require(stakeIndex < userStakes[msg.sender].length,
"Invalid stake index");
117             StakeInfo storage info = userStakes[msg.sender][stakeIndex];
118             require(!info.withdrawn, "Stake already withdrawn");
119
120             uint256 pending = calculateClaimableReward(info);
121             require(pending > 0, "No reward available");
122
123             // Update all reward tracking fields
124             info.claimedReward += pending;
125             info.rewardDebt = (info.stakingShare *
```

```
totalStakes.rewardIndex) / 1e18;
126
127             stakingToken.transfer(msg.sender, pending);
128             emit ClaimedReward(msg.sender, stakeIndex, pending);
129         }
131         function calculateClaimableReward(StakeInfo memory info) public
view returns (uint256) {
132             if (info.withdrawn)
133                 return 0;
134
135             uint256 accumulated = (info.stakingShare *
totalStakes.rewardIndex) / 1e18;
136             if (accumulated < info.rewardDebt) return 0;
137             return accumulated - info.rewardDebt;
138         }
170         function updateReward() external accrueReward {
171
172         }
184         function calculateAPR() public view returns(uint256) {
185             if (totalStakes.stakingShare == 0)
186                 return 0;
187
188             return getDistributionRate() * 365 * 1e18 /
totalStakes.stakingShare;
189         }
191         function getLockDuration(LockDuration duration) public pure
returns (uint256) {
192             if (duration == LockDuration.QUATERLY) return YEAR / 4;
193             if (duration == LockDuration.HALF_YEARLY) return YEAR / 2;
194             if (duration == LockDuration.YEARLY) return YEAR;
195             return 0;
196         }
198         function getLockMultiplier(LockDuration duration) public pure
returns (uint256) {
199             if (duration == LockDuration.QUATERLY) return 125;
200             if (duration == LockDuration.HALF_YEARLY) return 150;
201             if (duration == LockDuration.YEARLY) return 200;
202             return 100;
203         }
205         function getAmountMultiplier(uint256 amount) public pure
returns (uint256) {
206             if (amount >= 500_000 ether) return 250;
207             if (amount >= 200_000 ether) return 200;
208             if (amount >= 100_000 ether) return 150;
209             if (amount >= 50_000 ether) return 125;
210             if (amount >= 20_000 ether) return 115;
211             return 100;
212         }
214         function getStakes(address user) external view returns
(StakeInfo[] memory stakes) {
215             stakes = userStakes[user];
216             for (uint256 i = 0; i < stakes.length; ++i) {
217                 stakes[i].pendingReward =
calculateClaimableReward(stakes[i]);
218             }
219         }
```

```
221         function getTotalStakes() external view returns (TotalStakeInfo
memory) {
222             return totalStakes;
223         }
```

Consider adding formal verification proofs

Description:

Consider using formal verification to mathematically prove that your code does what is intended, and does not have any edge cases with unexpected behavior. The solidity compiler itself has this functionality built in based off of SMTChecker.

Severity:

NC

Snippet:

```
File: Various Files

None
```

Assembly code blocks should be thoroughly commented

Description:

In Solidity, assembly blocks are often harder to read and understand due to their low-level nature. Detailed comments can make the code's purpose and functionality clear, aiding in maintenance, debugging, and reviews. Moreover, comments can be crucial for auditors and other developers to understand the contract's security implications, reducing the risk of oversights and vulnerabilities.

Severity:

NC

Snippet:

```
File: @openzeppelin/contracts/utils/Strings.sol

50                  assembly ("memory-safe") {
51                      ptr := add(buffer, add(32, length))
52                  }
55                      assembly ("memory-safe") {
56                          mstore8(ptr, byte(mod(value, 10), HEX_DIGITS))
57                      }
116             assembly ("memory-safe") {
117                 hashValue := shr(96, keccak256(add(buffer, 0x22), 40))
118             }
470             assembly ("memory-safe") {
471                 mstore(output, outputLength)
472                 mstore(0x40, add(output, shl(5, shr(5,
add(outputLength, 63)))))
473             }
486             assembly ("memory-safe") {
487                 value := mload(add(buffer, add(0x20, offset)))
488             }
```

Large multiples of ten should use scientific notation (e.g. 1e6) rather than decimal literals (e.g. 1000000), for readability

Description:

Using scientific notation for large multiples of ten improves code readability. Instead of writing large decimal literals, consider using scientific notation.

Severity:

NC

Snippet:

```
File: contracts/TokenStaking.sol

41          uint256 constant totalRewardTokens = 100_000_000 ether;
208             if (amount >= 100_000 ether) return 150;
40          uint256 constant DISTRIBITE_UNIT = 10_000 ether;
```

Common functions should be refactored to a common base contract

Description:

The functions below have the same implementation as is seen in other files. The functions should be refactored into functions of a common base contract.

Severity:

NC

Snippet:

```
File: contracts/TeamAllocation.sol

/// @audit seen in ./contracts/KOLAllocation.sol
    function allocateTokens(
        address[] calldata _wallets,
        string[] calldata names,
        uint256[] calldata amounts
    ) external onlyOwner {
        require(
            _wallets.length == names.length && names.length ==
amounts.length,
            "Mismatched input lengths"
        );

        for (uint256 i = 0; i < _wallets.length; ++i) {
            wallets.push(_wallets[i]);
            membersInfo.push(Member({
                name: names[i],
                balance: amounts[i],
                startTime: block.timestamp,
                withdrawn: 0
            }));
            totalAllocated += amounts[i];
        }
    }
/// @audit seen in ./contracts/KOLAllocation.sol
    function withdraw() external {
        uint256 index = getMemberIndex(msg.sender);
        require(index != NOT_FOUND, "No Member wallet");

        Member storage member = membersInfo[index];
        uint256 withdrawable = calculateWithdrawable(member);
        require(withdrawable > 0, "Nothing to withdraw");

        member.withdrawn += withdrawable;
        totalWithdrawn += withdrawable;
        token.transfer(msg.sender, withdrawable);
```

```
    }
/// @audit seen in ./contracts/KOLAllocation.sol
    function getWithdrawable() external view returns (uint256) {
        uint256 index = getMemberIndex(msg.sender);
        require(index != NOT_FOUND, "No Member wallet");
        return calculateWithdrawable(membersInfo[index]);
    }
/// @audit seen in ./contracts/KOLAllocation.sol
    function calculateWithdrawable(Member memory member) internal view
returns (uint256) {
        if (block.timestamp < member.startTime + CLIFF_DURATION) {
            return 0;
        }

        uint256 elapsed = block.timestamp - member.startTime;
        uint256 vested;

        if (elapsed >= VESTING_DURATION) {
            vested = member.balance;
        } else {
            // 20% after cliff, remaining linearly over remaining time
            uint256 initial = member.balance / 5;
            uint256 linearPart = (member.balance * 4 * (elapsed -
CLIFF_DURATION)) / (VESTING_DURATION * 5);
            vested = initial + linearPart;
        }

        if (vested > member.balance) vested = member.balance;

        return vested - member.withdrawn;
    }
/// @audit seen in ./contracts/KOLAllocation.sol
    function getMemberIndex(address wallet) internal view returns (uint256)
{
        for (uint256 i = 0; i < wallets.length; ++i) {
            if (wallets[i] == wallet) return i;
        }
        return NOT_FOUND;
    }
/// @audit seen in ./contracts/KOLAllocation.sol
    function getMemberCount() external view returns (uint256) {
        return wallets.length;
    }
/// @audit seen in ./contracts/KOLAllocation.sol
    function getMember(address wallet) external view returns (Member
memory) {
        uint256 index = getMemberIndex(wallet);
        require(index != NOT_FOUND, "No Member wallet");
        return membersInfo[index];
    }
```

Polymorphic functions make security audits more time-consuming and error-prone

Description:

The instances below point to one of two functions with the same name. Consider naming each function differently, in order to make code navigation and analysis easier.

Severity:

NC

Snippet:

```
File: contracts/TokenPresale.sol

131         function buyTokens() public payable nonReentrant saleActive {
```

Event names should use CamelCase

Description:

According to the Solidity style guide event names should be in `CamelCase`.

Severity:

NC

Snippet:

```
File: @openzeppelin/contracts/interfaces/IERC5267.sol

10          event EIP712DomainChanged();
```

Consider using `SafeTransferLib.safeTransferETH()` or `Address.sendValue()` for clearer semantic meaning

Description:

These Functions indicate their purpose with their name more clearly than using low-level calls.

Severity:

NC

Snippet:

```
File: @openzeppelin/contracts-upgradeable/utils/AddressUpgradeable.sol

67              (bool success, ) = recipient.call{value: amount}("");
```

Error messages should descriptive, rather that cryptic

Description:

Severity:

NC

Snippet:

```
File: contracts/GasCreditVault.sol

107             require(!paused, "Paused");
115             require(paused, "Unpaused");
```

Missing timelock for critical parameter change

Description:

Timelocks prevent users from being surprised by changes.

Severity:

NC

Snippet:

```
File: contracts/TokenStaking.sol

68              stakingStartedAt = startAt;
```

Setters should prevent re-setting of the same value

Description:

This especially problematic when the setter also emits the same value, which may be confusing to offline parsers.

Severity:

NC

Snippet:

```
File: contracts/TokenStaking.sol

64          function setStartTimeForStaking(uint256 startAt) external
onlyOwner {
65              require(stakingStartedAt == 0, 'Staking already started');
66              require(startAt > block.timestamp, 'Invalid timestamp');
67
68              stakingStartedAt = startAt;
69          }
```

Consider splitting long calculations

Description:

The longer a string of operations is, the harder it is to understand it. Consider splitting the full calculation into more steps, with more descriptive temporary variable names, and add extensive comments.

Severity:

NC

Snippet:

```
File: contracts/TeamAllocation.sol

85                  uint256 linearPart = (member.balance * 4 * (elapsed -
CLIFF_DURATION)) / (VESTING_DURATION * 5);
```

High cyclomatic complexity

Description:

Consider breaking down these blocks into more manageable units, by splitting things into utility functions, by reducing nesting, and by using early returns.

Severity:

NC

Snippet:

```
File: @openzeppelin/contracts/utils/Strings.sol

446         function escapeJSON(string memory input) internal pure returns
(string memory) {
447             bytes memory buffer = bytes(input);
448             bytes memory output = new bytes(2 * buffer.length); //
worst case scenario
449             uint256 outputLength = 0;
450
451             for (uint256 i; i < buffer.length; ++i) {
452                 bytes1 char = bytes1(_unsafeReadBytesOffset(buffer, i));
453                 if (((SPECIAL_CHARS_LOOKUP & (1 << uint8(char))) != 0))
```

```
{
454                     output[outputLength++] = "\\";
455                     if (char == 0x08) output[outputLength++] = "b";
456                     else if (char == 0x09) output[outputLength++] = "t";
457                     else if (char == 0x0a) output[outputLength++] = "n";
458                     else if (char == 0x0c) output[outputLength++] = "f";
459                     else if (char == 0x0d) output[outputLength++] = "r";
460                     else if (char == 0x5c) output[outputLength++] = "\
\";
461                     else if (char == 0x22) {
462                         // solhint-disable-next-line quotes
463                         output[outputLength++] = '"';
464                     }
465                 } else {
466                     output[outputLength++] = char;
467                 }
468             }
469             // write the actual length and deallocate unused memory
470             assembly ("memory-safe") {
471                 mstore(output, outputLength)
472                 mstore(0x40, add(output, shl(5, shr(5,
add(outputLength, 63)))))
473             }
474
475             return string(output);
476         }
```

Use of override is unnecessary

Description:

Starting with Solidity version 0.8.8, using the override keyword when the function solely overrides an interface function, and the function doesn't exist in multiple base contracts, is unnecessary.

Severity:

NC

Snippet:

```
File: contracts/MetaTxGateway.sol

    function _authorizeUpgrade(address newImplementation) internal override
onlyOwner {}
```

Non-`external`/`public` function names should begin with an underscore

Description:

According to the Solidity Style Guide, non-`external`/`public` function names should begin with an underscore

Severity:

NC

Snippet:

```
File: contracts/TokenStaking.sol

140         function updateStakingShare(address user) internal accrueReward
{
175         function getDistributionRate() internal view returns(uint256) {
180         function calculateStakingShare(uint256 amount, LockDuration
duration) internal pure returns(uint256) {
```

Unused import

Description:

The identifier is imported but never used within the file

Severity:

NC

Snippet:

```
File: contracts/MetaTxGateway.sol

5       import {IERC20} from "node_modules/@openzeppelin/contracts/token/
ERC20/IERC20.sol";
```

Use `string.concat()` on strings instead of `abi.encodePacked()` for clearer semantic meaning

Description:

Starting with version 0.8.12, Solidity has the `string.concat()` function, which allows one to concatenate a list of strings, without extra padding. Using this function rather than `abi.encodePacked()` makes the intended operation more clear, leading to less reviewer confusion.

Severity:

NC

Snippet:

```
File: contracts/MetaTxGateway.sol

279                 keccak256(abi.encodePacked(metaTxHashes)),
394                 keccak256(abi.encodePacked(metaTxHashes)),
```

Unused function parameter

Description:

Comment out the variable name to suppress compiler warnings.

Severity:

NC

Snippet:

```
File: contracts/MetaTxGateway.sol

430         function _authorizeUpgrade(address newImplementation) internal
override onlyOwner {}
```

Unsafe conversion from unsigned to signed values

Description:

Solidity follows two's complement rules for its integers, meaning that the most significant bit for signed integers is used to denote the sign, and converting between the two requires inverting all of the bits and adding one. Because of this, casting an unsigned integer to a signed one may result in a change of the sign and or magnitude of the value. For example, `int8(type(uint8).max)` is not equal to `type(int8).max`, but is equal to `-1`. `type(uint8).max` in binary is `11111111`, which if cast to a signed value, means the first binary `1` indicates a negative value, and the binary `1`s, invert to all zeroes, and when one is added, it becomes one, but negative, and therefore the decimal value of binary `11111111` is `-1`.

Severity:

NC

Snippet:

```
File: @openzeppelin/contracts/utils/Strings.sol

280                 return (true, negativeSign ? -int256(absValue) :
int256(absValue));
```

Put all system-wide constants in one file

Description:

Putting all the system-wide constants in a single file improves code readability, makes it easier to understand the basic configuration and limitations of the system, and makes maintenance easier.

Severity:

NC

Snippet:

```
File: contracts/TeamAllocation.sol

17          uint256 public constant VESTING_DURATION = 180 days;

18          uint256 public constant CLIFF_DURATION = 30 days;
```

Add inline comments for unnamed variables

Description:

`function foo(address x, address)` -> `function foo(address x, address /* y */)`

Severity:

NC

Snippet:

```
File: contracts/mock/MockAggregatorV3.sol

27          function getRoundData(uint80)
28              external
29              view
30              override
31              returns (
32                  uint80 roundId,
33                  int256 answer,
34                  uint256 startedAt,
35                  uint256 updatedAt,
36                  uint80 answeredInRound
37              )
38          {
```

Consider adding emergency-stop functionality

Description:

Adding a way to quickly halt protocol functionality in an emergency, rather than having to pause individual contracts one-by-one, will make in-progress hack mitigation faster and much less stressful.

Severity:

NC

Snippet:

```
File: contracts/TokenStaking.sol

7       contract TokenStaking is Ownable {
```

Named imports of parent contracts are missing

Description:

Severity:

NC

Snippet:

```
File: contracts/TokenStaking.sol

7       contract TokenStaking is Ownable {
```

Use `bytes.concat()` on bytes instead of `abi.encodePacked()` for clearer semantic meaning

Description:

Starting with version 0.8.4, Solidity has the `bytes.concat()` function, which allows one to concatenate a list of bytes/strings, without extra padding. Using this function rather than `abi.encodePacked()` makes the intended operation more clear, leading to less reviewer confusion.

Severity:

NC

Snippet:

```
File: contracts/MetaTxGateway.sol

285             bytes32 digest = keccak256(abi.encodePacked(
286                 "\x19\x01",
287                 domainSeparator,
288                 structHash
289             ));
400             return keccak256(abi.encodePacked(
401                 "\x19\x01",
402                 domainSeparator,
403                 structHash
404             ));
```

Consider using `SafeTransferLib.safeTransferETH()` or `Address.sendValue()` for clearer semantic meaning

Description:

These Functions indicate their purpose with their name more clearly than using low-level calls.

Severity:

NC

Snippet:

```
File: contracts/MetaTxGateway.sol

231                     (bool refundSuccess, ) = payable(from).call{value:
refundAmount}("");
```

Style guide: State and local variables should be named using lowerCamelCase

Description:

The Solidity style guide says to use mixedCase for local and state variable names. Note that while OpenZeppelin may not follow this advice, it still is the recommended way of naming variables.

Severity:

NC

Snippet:

```
File: @openzeppelin/contracts-upgradeable/utils/ContextUpgradeable.sol

36          uint256[50] private __gap;
```

Pure function accesses storage

Description:

While the compiler currently flags functions like these as being pure, this is a bug which will be fixed in a future version, so it's best to not use pure visibility, in order to not break when this bug is fixed.

Severity:

NC

Snippet:

```
File: @openzeppelin/contracts/utils/StorageSlot.sol

66          function getAddressSlot(bytes32 slot) internal pure returns
(AddressSlot storage r) {
67              assembly ("memory-safe") {
68                  r.slot := slot
69              }
70          }
75          function getBooleanSlot(bytes32 slot) internal pure returns
(BooleanSlot storage r) {
```

```
76              assembly ("memory-safe") {
77                  r.slot := slot
78              }
79          }
84          function getBytes32Slot(bytes32 slot) internal pure returns
(Bytes32Slot storage r) {
85              assembly ("memory-safe") {
86                  r.slot := slot
87              }
88          }
93          function getUint256Slot(bytes32 slot) internal pure returns
(Uint256Slot storage r) {
94              assembly ("memory-safe") {
95                  r.slot := slot
96              }
97          }
102         function getInt256Slot(bytes32 slot) internal pure returns
(Int256Slot storage r) {
103             assembly ("memory-safe") {
104                 r.slot := slot
105             }
106         }
111         function getStringSlot(bytes32 slot) internal pure returns
(StringSlot storage r) {
112             assembly ("memory-safe") {
113                 r.slot := slot
114             }
115         }
120         function getStringSlot(string storage store) internal pure
returns (StringSlot storage r) {
121             assembly ("memory-safe") {
122                 r.slot := store.slot
123             }
124         }
129         function getBytesSlot(bytes32 slot) internal pure returns
(BytesSlot storage r) {
130             assembly ("memory-safe") {
131                 r.slot := slot
132             }
133         }
138         function getBytesSlot(bytes storage store) internal pure
returns (BytesSlot storage r) {
139             assembly ("memory-safe") {
140                 r.slot := store.slot
141             }
142         }
```

Unnecessary cast

Description:

The variable is being cast to its own type

Severity:

NC

Snippet:

```
File: @openzeppelin/contracts/utils/ShortStrings.sol

57              return ShortString.wrap(bytes32(uint256(bytes32(bstr)) |
bstr.length));
```

Unusual loop variable

Description:

The normal name for loop variables is i, and when there is a nested loop, to use j. Not following this convention may lead to some reviewer confusion.

Severity:

NC

Snippet:

```
File: @openzeppelin/contracts/utils/Arrays.sol

124                 for (uint256 it = begin + 0x20; it < end; it += 0x20) {
125                     if (comp(_mload(it), pivot)) {
126                         // If the value stored at the iterator's
position comes before the pivot, we increment the
127                         // position of the pivot and move the value
there.
128                         pos += 0x20;
129                         _swap(pos, it);
130                     }
131                 }
```

Use the latest solidity (prior to 0.8.20 if on L2s) for deployment

Description:

Since deployed contracts should not use floating pragmas, I've flagged all instances where a version prior to 0.8.19 is allowed by the version pragma

Severity:

NC

Snippet:

```
File: @openzeppelin/contracts-upgradeable/utils/StorageSlotUpgradeable.sol

5       pragma solidity ^0.8.0;
```

Events should use parameters to convey information

Description:

For example, rather than using event Paused() and event Unpaused(), use event PauseState(address indexed whoChangedIt, bool wasPaused, bool isNowPaused)

Severity:

NC

Snippet:

```
File: contracts/GasCreditVault.sol

200             emit Paused();
211             emit Unpaused();
```

Visibility should be set explicitly rather than defaulting to internal

Description:

Visibility of state variables should be set explicitly rather than defaulting to internal.

Severity:

NC

Snippet:

```
File: contracts/TokenStaking.sol

39          uint256 constant YEAR = 365 days;
40          uint256 constant DISTRIBITE_UNIT = 10_000 ether;
41          uint256 constant totalRewardTokens = 100_000_000 ether;
```

Event declarations should have NatSpec @param annotations

Description:

Documents a parameter just like in Doxygen (must be followed by parameter name)

Severity:

NC

Snippet:

```
File: contracts/TokenStaking.sol

50          event RewardMultiplierUpdated(uint256);
51          event Staked(address indexed user, uint256 stakedAmount,
LockDuration duration);
52          event Withdrawn(address indexed user, uint256 stakeIndex,
uint256 amount);
53          event ClaimedReward(address indexed user, uint256 stakeIndex,
```

```
uint256 reward);
```

Event declarations should have NatSpec @dev annotations

Description:

Explain to a developer any extra details

Severity:

NC

Snippet:

```
File: contracts/TokenStaking.sol

50          event RewardMultiplierUpdated(uint256);
51          event Staked(address indexed user, uint256 stakedAmount,
LockDuration duration);
52          event Withdrawn(address indexed user, uint256 stakeIndex,
uint256 amount);
53          event ClaimedReward(address indexed user, uint256 stakeIndex,
uint256 reward);
```

Function definitions should have NatSpec @dev annotations

Description:

Explain to a developer any extra details

Severity:

NC

Snippet:

```
File: contracts/TokenStaking.sol

55          constructor(address _stakingToken, address _owner)
Ownable(_owner) {
```

```
59          function updateDistributeRate(uint256 newMultiplier) external
onlyOwner accrueReward {
64          function setStartTimeForStaking(uint256 startAt) external
onlyOwner {
77          function stake(uint256 amount, LockDuration duration) external
stakingActive {
99          function withdraw(uint256 stakeIndex) external {
115         function claimReward(uint256 stakeIndex) public accrueReward {
131         function calculateClaimableReward(StakeInfo memory info) public
view returns (uint256) {
140         function updateStakingShare(address user) internal accrueReward
{
170         function updateReward() external accrueReward {
175         function getDistributionRate() internal view returns(uint256) {
180         function calculateStakingShare(uint256 amount, LockDuration
duration) internal pure returns(uint256) {
184         function calculateAPR() public view returns(uint256) {
191         function getLockDuration(LockDuration duration) public pure
returns (uint256) {
198         function getLockMultiplier(LockDuration duration) public pure
returns (uint256) {
205         function getAmountMultiplier(uint256 amount) public pure
returns (uint256) {
214         function getStakes(address user) external view returns
(StakeInfo[] memory stakes) {
221         function getTotalStakes() external view returns (TotalStakeInfo
memory) {
```

Function definitions should have NatSpec @notice annotations

Description:

Explain to an end user what this does

Severity:

NC

Snippet:

```
File: contracts/TokenStaking.sol

55          constructor(address _stakingToken, address _owner)
Ownable(_owner) {
59          function updateDistributeRate(uint256 newMultiplier) external
onlyOwner accrueReward {
64          function setStartTimeForStaking(uint256 startAt) external
onlyOwner {
```

```
77          function stake(uint256 amount, LockDuration duration) external
stakingActive {
99          function withdraw(uint256 stakeIndex) external {
115         function claimReward(uint256 stakeIndex) public accrueReward {
131         function calculateClaimableReward(StakeInfo memory info) public
view returns (uint256) {
140         function updateStakingShare(address user) internal accrueReward
{
170         function updateReward() external accrueReward {
175         function getDistributionRate() internal view returns(uint256) {
180         function calculateStakingShare(uint256 amount, LockDuration
duration) internal pure returns(uint256) {
184         function calculateAPR() public view returns(uint256) {
191         function getLockDuration(LockDuration duration) public pure
returns (uint256) {
198         function getLockMultiplier(LockDuration duration) public pure
returns (uint256) {
205         function getAmountMultiplier(uint256 amount) public pure
returns (uint256) {
214         function getStakes(address user) external view returns
(StakeInfo[] memory stakes) {
221         function getTotalStakes() external view returns (TotalStakeInfo
memory) {
```

Interface declarations should have NatSpec @title annotations

Description:

A title that should describe the contract/interface

Severity:

NC

Snippet:

```
File: contracts/TokenPresale.sol

8       interface AggregatorV3Interface {
```

Interface declarations should have NatSpec @author annotations

Description:

The name of the author

Severity:

NC

Snippet:

```
File: contracts/TokenPresale.sol

8       interface AggregatorV3Interface {
```

Interface declarations should have NatSpec @notice annotations

Description:

Explain to an end user what this does

Severity:

NC

Snippet:

```
File: contracts/TokenPresale.sol

8       interface AggregatorV3Interface {
```

Interface declarations should have NatSpec @dev annotations

Description:

Explain to a developer any extra details

Severity:

NC

Snippet:

```
File: contracts/TokenPresale.sol

8       interface AggregatorV3Interface {
```

Abstract contract declarations should have NatSpec @title annotations

Description:

A title that should describe the contract/interface

Severity:

NC

Snippet:

```
File: @openzeppelin/contracts/utils/ReentrancyGuard.sol

25      abstract contract ReentrancyGuard {
```

Abstract contract declarations should have NatSpec @author annotations

Description:

The name of the author

Severity:

NC

Snippet:

```
File: @openzeppelin/contracts/utils/ReentrancyGuard.sol

25      abstract contract ReentrancyGuard {
```

Abstract contract declarations should have NatSpec @notice annotations

Description:

Explain to an end user what this does

Severity:

NC

Snippet:

```
File: @openzeppelin/contracts/utils/ReentrancyGuard.sol

25      abstract contract ReentrancyGuard {
```

Library declarations should have Natspec @title annotations

Description:

A title that should describe the contract/interface

Severity:

NC

Snippet:

```
File: @openzeppelin/contracts/utils/Strings.sol

13      library Strings {
```

Library declarations should have Natspec @author annotations

Description:

The name of the author

Severity:

NC

Snippet:

```
File: @openzeppelin/contracts/utils/Strings.sol

13      library Strings {
```

Library declarations should have Natspec @notice annotations

Description:

Explain to an end user what this does

Severity:

NC

Snippet:

```
File: @openzeppelin/contracts/utils/Strings.sol

13      library Strings {
```

Library declarations should have Natspec @dev annotations

Description:

Explain to a developer any extra details

Severity:

NC

Snippet:

```
File: @openzeppelin/contracts/utils/Panic.sol

26      library Panic {
```

Modifier definitions should have Natspec @notice annotations

Description:

Explain to an end user what this does

Severity:

NC

Snippet:

```
File: contracts/TokenStaking.sol

71          modifier stakingActive() {
152         modifier accrueReward() {
```

Modifier definitions should have Natspec @dev annotations

Description:

Explain to a developer any extra details

Severity:

NC

Snippet:

```
File: contracts/TokenStaking.sol

71          modifier stakingActive() {
152         modifier accrueReward() {
```

Contract definitions should have Natspec @title annotations

Description:

title that should describe the contract

Severity:

NC

Snippet:

```
File: contracts/TokenStaking.sol

7       contract TokenStaking is Ownable {
```

Contract definitions should have Natspec @author annotations

Description:

The name of the author

Severity:

NC

Snippet:

```
File: contracts/TokenStaking.sol

7       contract TokenStaking is Ownable {
```

Contract definitions should have Natspec @notice annotations

Description:

Explain to an end user what this does

Severity:

NC

Snippet:

```
File: contracts/TokenStaking.sol

7       contract TokenStaking is Ownable {
```

Contract definitions should have Natspec @dev annotations

Description:

Explain to a developer any extra details

Severity:

NC

Snippet:

```
File: contracts/TokenStaking.sol

7       contract TokenStaking is Ownable {
```

Event definitions should have Natspec @notice annotations

Description:

Explain to an end user what this does

Severity:

NC

Snippet:

```
File: contracts/TokenStaking.sol

50          event RewardMultiplierUpdated(uint256);
51          event Staked(address indexed user, uint256 stakedAmount,
LockDuration duration);
52          event Withdrawn(address indexed user, uint256 stakeIndex,
uint256 amount);
53          event ClaimedReward(address indexed user, uint256 stakeIndex,
uint256 reward);
```

State variable declarations should have Natspec @notice annotations

Description:

Explain to an end user what this does

Severity:

NC

Snippet:

```
File: contracts/TokenStaking.sol

8           IERC20 public immutable stakingToken;
39          uint256 constant YEAR = 365 days;
40          uint256 constant DISTRIBITE_UNIT = 10_000 ether;
41          uint256 constant totalRewardTokens = 100_000_000 ether;
43          mapping(address => StakeInfo[]) public userStakes;
44          TotalStakeInfo public totalStakes;
```

```
46          uint256 public rewardMultiplier = 1e18;
47          uint256 public stakingStartedAt;
48          bool public stakingEnded;
```

State variable declarations should have Natspec @dev annotations

Description:

Explain to a developer any extra details

Severity:

NC

Snippet:

```
File: contracts/TokenStaking.sol

8           IERC20 public immutable stakingToken;
39          uint256 constant YEAR = 365 days;
40          uint256 constant DISTRIBITE_UNIT = 10_000 ether;
41          uint256 constant totalRewardTokens = 100_000_000 ether;
43          mapping(address => StakeInfo[]) public userStakes;
44          TotalStakeInfo public totalStakes;
46          uint256 public rewardMultiplier = 1e18;
47          uint256 public stakingStartedAt;
48          bool public stakingEnded;
```

Functions should have Natspec @return annotations

Description:

Documents the return variables of a contract’s function

Severity:

NC

Snippet:

```
File: contracts/TokenStaking.sol

131         function calculateClaimableReward(StakeInfo memory info) public
view returns (uint256) {
175         function getDistributionRate() internal view returns(uint256) {
180         function calculateStakingShare(uint256 amount, LockDuration
duration) internal pure returns(uint256) {
184         function calculateAPR() public view returns(uint256) {
191         function getLockDuration(LockDuration duration) public pure
returns (uint256) {
198         function getLockMultiplier(LockDuration duration) public pure
returns (uint256) {
205         function getAmountMultiplier(uint256 amount) public pure
returns (uint256) {
214         function getStakes(address user) external view returns
(StakeInfo[] memory stakes) {
221         function getTotalStakes() external view returns (TotalStakeInfo
memory) {
```

Functions should have Natspec @param annotations

Description:

Documents a parameter just like in Doxygen (must be followed by parameter name)

Severity:

NC

Snippet:

```
File: contracts/TokenStaking.sol

55          constructor(address _stakingToken, address _owner)
Ownable(_owner) {
59          function updateDistributeRate(uint256 newMultiplier) external
onlyOwner accrueReward {
64          function setStartTimeForStaking(uint256 startAt) external
onlyOwner {
77          function stake(uint256 amount, LockDuration duration) external
stakingActive {
99          function withdraw(uint256 stakeIndex) external {
115         function claimReward(uint256 stakeIndex) public accrueReward {
```

```
131         function calculateClaimableReward(StakeInfo memory info) public
view returns (uint256) {
140         function updateStakingShare(address user) internal accrueReward
{
170         function updateReward() external accrueReward {
175         function getDistributionRate() internal view returns(uint256) {
180         function calculateStakingShare(uint256 amount, LockDuration
duration) internal pure returns(uint256) {
184         function calculateAPR() public view returns(uint256) {
191         function getLockDuration(LockDuration duration) public pure
returns (uint256) {
198         function getLockMultiplier(LockDuration duration) public pure
returns (uint256) {
205         function getAmountMultiplier(uint256 amount) public pure
returns (uint256) {
214         function getStakes(address user) external view returns
(StakeInfo[] memory stakes) {
221         function getTotalStakes() external view returns (TotalStakeInfo
memory) {
```

Missing events in sensitive functions

Description:

Sensitive setter functions in smart contracts often alter critical state variables. Without events emitted in these functions, external observers or dApps cannot easily track or react to these state changes. Missing events can obscure contract activity, hampering transparency and making integration more challenging. To resolve this, incorporate appropriate event emissions within these functions. Events offer an efficient way to log crucial changes, aiding in real-time tracking and post-transaction verification..

Severity:

NC

Snippet:

```
File: contracts/TokenStaking.sol

64          function setStartTimeForStaking(uint256 startAt) external
onlyOwner {
65              require(stakingStartedAt == 0, 'Staking already started');
66              require(startAt > block.timestamp, 'Invalid timestamp');
67
68              stakingStartedAt = startAt;
69          }
```

If statement control structures do not comply with best practices

Description:

If statements which include a single line do not need to have curly brackets, however according to the Solidiity style guide the line of code executed upon the if statement condition being met should still be on the next line, not on the same line as the if statement declaration.

Severity:

NC

Snippet:

```
File: contracts/TokenStaking.sol

136             if (accumulated < info.rewardDebt) return 0;
192             if (duration == LockDuration.QUATERLY) return YEAR / 4;
193             if (duration == LockDuration.HALF_YEARLY) return YEAR / 2;
194             if (duration == LockDuration.YEARLY) return YEAR;
199             if (duration == LockDuration.QUATERLY) return 125;
200             if (duration == LockDuration.HALF_YEARLY) return 150;
201             if (duration == LockDuration.YEARLY) return 200;
206             if (amount >= 500_000 ether) return 250;
207             if (amount >= 200_000 ether) return 200;
208             if (amount >= 100_000 ether) return 150;
209             if (amount >= 50_000 ether) return 125;
210             if (amount >= 20_000 ether) return 115;
```

A event should be emitted if a non immutable state variable is set in a constructor

Description:

Severity:

NC

Snippet:

```
File: contracts/DIVote.sol

32          constructor(ERC20Votes _voteToken, address _owner)
Ownable(_owner) {
33              voteToken = _voteToken;
34          }
```

Unused file

Description:

The file is never imported by any other source file. If the file is needed for tests, it should be moved to a test directory

Severity:

NC

Snippet:

```
File: contracts/utils/Strings.sol

/// @auditbase `@openzeppelin/contracts/utils/Strings.sol` not used in any
contracts
/// @auditbase `@openzeppelin/contracts/utils/Strings.sol` not used in any
contracts
```

Public state arrays should have a getter to return all elements

Description:

In Solidity, public state variables automatically generate a getter function. For non-array types, this is straightforward: it simply returns the value. However, for arrays, the automatically generated getter only allows retrieval of an element at a specific index, not the entire array. This is mainly to prevent unintentional high gas costs, as returning the entire array can be expensive if it's large. If developers want to retrieve the whole array, they must explicitly define a function, as auto-generation could inadvertently expose contracts to gas-related vulnerabilities or lead to unwanted behavior for larger arrays.

Severity:

NC

Snippet:

```
File: contracts/TeamAllocation.sol

21          address[] public wallets;
22          Member[] public membersInfo;
```

It is best practice to use linear inheritance

Description:

In Solidity, complex inheritance structures can obfuscate code understanding, introducing potential security risks. Multiple inheritance, especially with overlapping function names or state variables, can cause unintentional overrides or ambiguous behavior. Resolution: Strive for linear and simple inheritance chains. Avoid diamond or circular inheritance patterns. Clearly document the purpose and relationships of base contracts, ensuring that overrides are intentional. Tools like Remix or Hardhat can visualize inheritance chains, assisting in verification. Keeping inheritance streamlined aids in better code readability, reduces potential errors, and ensures smoother audits and upgrades.

Severity:

NC

Snippet:

```
File: contracts/TokenPresale.sol

15      contract TokenPresale is Ownable, ReentrancyGuard {
```

Consider only defining one library/interface/ contract per sol file

Description:

Combining multiple libraries, interfaces, or contracts in a single file can lead to clutter, reduced readability, and versioning issues. **Resolution**: Adopt the best practice of defining only one library, interface, or contract per Solidity file. This modular approach enhances clarity, simplifies unit testing, and streamlines code review. Furthermore, segregating components makes version management easier, as updates to one component won't necessitate changes to a file housing multiple unrelated components. Structured file management can further assist in avoiding naming collisions and ensure smoother integration into larger systems or DApps.

Severity:

NC

Snippet:

```
File: contracts/TokenPresale.sol

8       interface AggregatorV3Interface {
9           function latestRoundData()
10              external
11              view
12              returns (uint80, int256 answer, uint256, uint256, uint80);
13      }
14
15      contract TokenPresale is Ownable, ReentrancyGuard {
```

Use a struct to encapsulate multiple function parameters

Description:

Using a struct to encapsulate multiple parameters in Solidity functions can significantly enhance code readability and maintainability. Instead of passing a long list of arguments, which can be error-prone and hard to manage, a struct allows grouping related data into a single, coherent entity. This approach simplifies function signatures and makes the code more organized. It also enhances code clarity, as developers can easily understand the relationship between the parameters. Moreover, it aids in future code modifications and expansions, as adding or modifying a parameter only requires changes in the struct definition, rather than in every function that uses these parameters.

Severity:

NC

Snippet:

```
File: contracts/MetaTxGateway.sol

190         function executeMetaTransactions(
191             address from,
192             MetaTransaction[] calldata metaTxs,
193             bytes calldata signature,
194             uint256 nonce,
195             uint256 deadline
196         ) external payable nonReentrant whenNotPaused returns (bool[]
memory successes) {
255         function _verifySignature(
256             address from,
257             MetaTransaction[] calldata metaTxs,
258             bytes calldata signature,
259             uint256 nonce,
260             uint256 deadline
261         ) internal view returns (bool valid) {
```

Avoid defining a function in a single line including it's contents

Description:

Severity:

NC

Snippet:

```
File: contracts/MetaTxGateway.sol

430         function _authorizeUpgrade(address newImplementation) internal
override onlyOwner {}
```

Empty bytes check is missing

Description:

When developing smart contracts in Solidity, it's crucial to validate the inputs of your functions. This includes ensuring that the bytes parameters are not empty, especially when they represent crucial data such as addresses, identifiers, or raw data that the contract needs to process.Missing empty bytes checks can lead to unexpected behaviour in your contract.For instance, certain operations might fail, produce incorrect results, or consume unnecessary gas when performed with empty bytes. Moreover, missing input validation can potentially expose your contract to malicious activity, including exploitation of unhandled edge cases.To mitigate these issues, always validate that bytes parameters are not empty when the logic of your contract requires it.

Severity:

NC

Snippet:

```
File: contracts/MetaTxGateway.sol

163         function _safeExecuteCall(address target, uint256 value, bytes
calldata data) external returns (bool success) {
190         function executeMetaTransactions(
```

```
191             address from,
192             MetaTransaction[] calldata metaTxs,
193             bytes calldata signature,
194             uint256 nonce,
195             uint256 deadline
196         ) external payable nonReentrant whenNotPaused returns (bool[]
memory successes) {
255         function _verifySignature(
256             address from,
257             MetaTransaction[] calldata metaTxs,
258             bytes calldata signature,
259             uint256 nonce,
260             uint256 deadline
261         ) internal view returns (bool valid) {
```

Defining All External/Public Functions in Contract Interfaces

Description:

It is preferable to have all the external and public function in an interface to make using them easier by developers. This helps ensure the whole API is extracted in a interface.

Severity:

NC

Snippet:

```
File: contracts/TokenStaking.sol

115         function claimReward(uint256 stakeIndex) public accrueReward {
131         function calculateClaimableReward(StakeInfo memory info) public
view returns (uint256) {
184         function calculateAPR() public view returns(uint256) {
191         function getLockDuration(LockDuration duration) public pure
returns (uint256) {
198         function getLockMultiplier(LockDuration duration) public pure
returns (uint256) {
205         function getAmountMultiplier(uint256 amount) public pure
returns (uint256) {
```

Don't Initialize Variables with Default Value

Description:

Severity:

G

Snippet:

```
File: contracts/TokenStaking.sol

for (uint256 i = 0; i < stakes.length; ++i) {
```

Cache Array Length Outside of Loop

Description:

If not cached, the solidity compiler will always read the length of the array during each iteration. That is, if it is a storage array, this is an extra sload operation (100 additional extra gas for each iteration except for the first) and if it is a memory array, this is an extra mload operation (3 additional gas for each iteration except for the first).

Severity:

G

Snippet:

```
File: contracts/TokenStaking.sol

216             for (uint256 i = 0; i < stakes.length; ++i) {
```

Use != 0 instead of > 0 for Unsigned Integer Comparison

Description:

Checking for `!= 0` is cheaper than `> 0` for unsigned integers.

Severity:

G

Snippet:

```
File: contracts/TokenStaking.sol

78              require(amount > 0, "Amount must be > 0");
78              require(amount > 0, "Amount must be > 0");
121             require(pending > 0, "No reward available");
121             require(pending > 0, "No reward available");
```

Using `private` rather than `public` for constants, saves gas

Description:

If needed, the values can be read from the verified contract source code, or if there are multiple values there can be a single getter function that returns a tuple of the values of all currently-public constants. Saves **3406-3606 gas** in deployment gas due to the compiler not having to create non-payable getter functions for deployment calldata, not having to store the bytes of the value outside of where it's used, and not adding another entry to the method ID table

Severity:

G

Snippet:

```
File: contracts/TeamAllocation.sol

uint256 public constant VESTING_DURATION = 180 days;
uint256 public constant CLIFF_DURATION = 30 days;
```

Long Revert Strings

Description:

Severity:

G

Snippet:

```
File: contracts/DIVote.sol

require(weight > 0, "No voting power at proposal start");
```

Use Custom Errors

Description:

Instead of using error strings, to reduce deployment and runtime cost, you should use Custom Errors. This would save both deployment and runtime cost.

Source: https://consensys.net/diligence/blog/2019/09/stop-using-string-error-messages/

Severity:

G

Snippet:

```
File: contracts/TokenStaking.sol

65              require(stakingStartedAt == 0, 'Staking already started');
66              require(startAt > block.timestamp, 'Invalid timestamp');
72              require(stakingStartedAt != 0 && block.timestamp >=
stakingStartedAt, 'Staking not started');
73              require(!stakingEnded, 'Staking ended');
78              require(amount > 0, "Amount must be > 0");
101             require(!info.withdrawn, "Already withdrawn");
102             require(block.timestamp >= info.startTime +
getLockDuration(info.lockDuration) || stakingEnded, "Lock not expired");
116             require(stakeIndex < userStakes[msg.sender].length,
"Invalid stake index");
118             require(!info.withdrawn, "Stake already withdrawn");
```

```
121             require(pending > 0, "No reward available");
```

++i costs less gas than i++, especially when it's used in for-loops (--i/i-- too)

Description:

Using ++i or --i instead of i++ or i-- can save gas, especially in for-loops.

Severity:

G

Snippet:

```
File: contracts/MetaTxGateway.sol

175             for (uint256 i = 0; i < metaTxs.length; i++) {
218             for (uint256 i = 0; i < metaTxs.length; i++) {
266             for (uint256 i = 0; i < metaTxs.length; i++) {
381             for (uint256 i = 0; i < metaTxs.length; i++) {
```

Use assembly to check for `address(0)`

Description:

Saves 6 gas per instance

Severity:

G

Snippet:

```
File: contracts/TokenAirdrop.sol

20              require(_token != address(0), "Invalid token address");
44              require(merkleRoot != bytes32(0), "Merkle root not set");
```

`internal` functions not called by the contract should be removed to save deployment gas

Description:

If the functions are required by an interface, the contract should inherit from that interface and use the override keyword

Severity:

G

Snippet:

```
File: contracts/MetaTxGateway.sol

    function _authorizeUpgrade(address newImplementation) internal override
onlyOwner {}
```

Use calldata instead of memory for function arguments that do not get mutated

Description:

Mark data types as `calldata` instead of `memory` where possible. This makes it so that the data is not automatically loaded into memory. If the data passed into the function does not need to be changed (like updating values in an array), it can be passed in as `calldata`. The one exception to this is if the argument must later be passed into another function that takes an argument that specifies `memory` storage.

Severity:

G

Snippet:

```
File: contracts/TokenStaking.sol

131         function calculateClaimableReward(StakeInfo memory info) public
view returns (uint256) {
```

Multiple address/ID mappings can be combined into a single mapping of an address/ID to a struct, where appropriate

Description:

Saves a storage slot for the mapping. Depending on the circumstances and sizes of types, can avoid a Gsset (20000 gas) per mapping combined. Reads and subsequent writes can also be cheaper when a function requires both values and they both fit in the same storage slot. Finally, if both fields are accessed in the same function, can save ~42 gas per access due to not having to recalculate the key's keccak256 hash (Gkeccak256 - 30 gas) and that calculation's associated stack operations.

Severity:

G

Snippet:

```
File: contracts/GasCreditVault.sol

55          mapping(address => TokenInfo) public tokenInfo;
56          mapping(address => uint256) public credits;
57          mapping(address => mapping(address => uint256)) public
creditsInToken;
```

Using storage instead of `memory` for structs/ arrays saves gas

Description:

When fetching data from a storage location, assigning the data to a memory variable causes all fields of the struct/array to be read from storage, which incurs a Gcoldsload (2100 gas) for each field of the struct/array. If the fields are read from the new memory variable, they incur an additional MLOAD rather than a cheap stack read. Instead of declearing the variable with the memory keyword, declaring the variable with the storage keyword and caching any fields that need to be re-read in stack variables, will be much cheaper, only incuring the Gcoldsload for the fields actually read. The only time it makes sense to read the whole struct/array into a memory variable, is if the full struct/

array is being returned by the function, is being passed to a function that requires memory, or if the array/struct is being read from another memory array/struct

Severity:

G

Snippet:

```
File: contracts/TokenPresale.sol

107             PaymentToken memory info = paymentTokens[payToken];
```

Multiple accesses of a mapping/array should use a local variable cache.

Description:

The instances below point to the second+ access of a value inside a mapping/array, within a function. Caching a mapping's value in a local storage or calldata variable when the value is accessed multiple times, saves ~42 gas per access due to not having to recalculate the key's keccak256 hash (Gkeccak256 - 30 gas) and that calculation's associated stack operations. Caching an array's struct avoids recalculating the array offsets into memory/calldata

Severity:

G

Snippet:

```
File: contracts/TokenStaking.sol

217                 stakes[i].pendingReward =
calculateClaimableReward(stakes[i]);
```

Internal functions only called once can be inlined to save gas

Description:

Not inlining costs 20 to 40 gas because of two extra JUMP instructions and additional stack operations needed for function calls.

Severity:

G

Snippet:

```
File: contracts/TokenStaking.sol

140         function updateStakingShare(address user) internal accrueReward
{
180         function calculateStakingShare(uint256 amount, LockDuration
duration) internal pure returns(uint256) {
```

Add unchecked {} for subtractions where the operands cannot underflow because of a previous require() or if-statement

Description:

`require(a <= b); x = b - a` => `require(a <= b); unchecked { x = b - a }`

Severity:

G

Snippet:

```
File: contracts/GasCreditVault.sol

429             require(block.timestamp - updatedAt <= PRICE_FEED_TIMEOUT,
"Price feed too stale");
459             require(block.timestamp - updatedAt <= PRICE_FEED_TIMEOUT,
"Price feed too stale");
```

Optimize names to save gas

Description:

public/external function names and public member variable names can be optimized to save gas.

Severity:

G

Snippet:

```
File: contracts/TokenStaking.sol

7       contract TokenStaking is Ownable {
```

Division by two should use bit shifting

Description:

/ 2 is the same as >> 1. While the compiler uses the SHR opcode to accomplish both, the version that uses division incurs an overhead of 20 gas due to JUMPs to and from a compiler utility function that introduces checks which can be avoided by using unchecked {} around the division by two

Severity:

G

Snippet:

```
File: contracts/TokenStaking.sol

193             if (duration == LockDuration.HALF_YEARLY) return YEAR / 2;
```

The result of function calls should be cached rather than re-calling the function

Description:

Caching the result of a function call in a local variable when the function is called multiple times can save gas due to avoiding the need to execute the function code multiple times.

Severity:

G

Snippet:

```
File: @openzeppelin/contracts/governance/utils/Votes.sol

186                 _push(_totalCheckpoints, _subtract,
SafeCast.toUint208(amount));
208                         SafeCast.toUint208(amount)
```

Splitting require() statements that use && saves gas

Description:

Splitting require statements that use && operator can save gas. There is a larger deployment gas cost, but with enough runtime calls, the change ends up being cheaper by 3 gas.

Severity:

G

Snippet:

```
File: contracts/TokenStaking.sol

72              require(stakingStartedAt != 0 && block.timestamp >=
stakingStartedAt, 'Staking not started');
```

Stack variable used as a cheaper cache for a state variable is only used once

Description:

If the variable is only accessed once, it's cheaper to use the state variable directly that one time, and save the 3 gas the extra stack assignment would spend.

Severity:

G

Snippet:

```
File: contracts/TokenPresale.sol

111             tokenAmount = (convertDecimals(amountIn, info.decimals, 18)
* tokenUSDPrice * 1e18) / (dynamicRate * 1e8); // Normalize decimals
```

Functions guaranteed to revert when called by normal users can be marked payable

Description:

If a function modifier such as onlyOwner is used, the function will revert if a normal user tries to pay the function. Marking the function as payable will lower the gas cost for legitimate callers because the compiler will not include checks for whether a payment was provided. The extra opcodes avoided are CALLVALUE(2),DUP1(3),ISZERO(3),PUSH2(3),JUMPI(10),PUSH1(3),DUP1(3),REVERT(0),JUMPDEST(1) which costs an average of about 21 gas per call to the function, in addition to the extra deployment cost.

Severity:

G

Snippet:

```
File: contracts/TokenStaking.sol

59          function updateDistributeRate(uint256 newMultiplier) external
```

```
onlyOwner accrueReward {
64          function setStartTimeForStaking(uint256 startAt) external
onlyOwner {
```

`<x> += <y>` costs more gas than `<x> = <x> + <y>` for state variables

Description:

Using the addition operator instead of plus-equals saves 113 gas

Severity:

G

Snippet:

```
File: contracts/TokenStaking.sol

107             totalStakes.totalStaked -= info.amount;
108             totalStakes.stakingShare -= info.stakingShare;
147             totalStakes.totalStaked += userStake.amount;
148             totalStakes.stakingShare += share;
163                 totalStakes.rewardIndex += (reward * 1e18) /
totalStakes.stakingShare;
165                 totalStakes.totalRewardDistributed += reward;
```

Constructors can be marked payable

Description:

Payable functions cost less gas to execute, since the compiler does not have to add extra checks to ensure that a payment wasn't provided. A constructor can safely be marked as payable, since only the deployer would be able to pass funds, and the project itself would not pass any funds.

Severity:

G

Snippet:

```
File: contracts/TokenStaking.sol

55          constructor(address _stakingToken, address _owner)
Ownable(_owner) {
56              stakingToken = IERC20(_stakingToken);
57          }
```

Remove unused local variables

Description:

Removing unused local variables saves gas.

Severity:

G

Snippet:

```
File: @openzeppelin/contracts/utils/Arrays.sol

34              function(uint256, uint256) pure returns (bool) comp
34              function(uint256, uint256) pure returns (bool) comp
34              function(uint256, uint256) pure returns (bool) comp
63              function(address, address) pure returns (bool) comp
63              function(address, address) pure returns (bool) comp
63              function(address, address) pure returns (bool) comp
92              function(bytes32, bytes32) pure returns (bool) comp
92              function(bytes32, bytes32) pure returns (bool) comp
92              function(bytes32, bytes32) pure returns (bool) comp
115         function _quickSort(uint256 begin, uint256 end,
function(uint256, uint256) pure returns (bool) comp) private pure {
115         function _quickSort(uint256 begin, uint256 end,
function(uint256, uint256) pure returns (bool) comp) private pure {
115         function _quickSort(uint256 begin, uint256 end,
function(uint256, uint256) pure returns (bool) comp) private pure {
195             function(address, address) pure returns (bool) input
195             function(address, address) pure returns (bool) input
195             function(address, address) pure returns (bool) input
196         ) private pure returns (function(uint256, uint256) pure returns
(bool) output) {
196         ) private pure returns (function(uint256, uint256) pure returns
(bool) output) {
196         ) private pure returns (function(uint256, uint256) pure returns
(bool) output) {
204             function(bytes32, bytes32) pure returns (bool) input
```

```
204             function(bytes32, bytes32) pure returns (bool) input
204             function(bytes32, bytes32) pure returns (bool) input
205         ) private pure returns (function(uint256, uint256) pure returns
(bool) output) {
205         ) private pure returns (function(uint256, uint256) pure returns
(bool) output) {
205         ) private pure returns (function(uint256, uint256) pure returns
(bool) output) {
```

`>=` costs less gas than `>`

Description:

The compiler uses opcodes `GT` and `ISZERO` for solidity code that uses `>`, but only requires `LT` for `>=`, which saves 3 gas.

Severity:

G

Snippet:

```
File: contracts/GasCreditVault.sol

305             uint256 creditCost = usdValue > minimumConsume ? usdValue :
minimumConsume;
```

Use solidity version 0.8.20 or above to improve gas performance

Description:

Upgrade to the latest solidity version 0.8.20 to get additional gas savings. See the latest release for reference: https://blog.soliditylang.org/2023/05/10/solidity-0.8.20-release-announcement/

Severity:

G

Snippet:

```
File: contracts/TokenStaking.sol

2       pragma solidity ^0.8.20;
```

Expression `is cheaper than new bytes(0)`

Description:

Severity:

G

Snippet:

```
File: @openzeppelin/contracts/utils/math/Math.sol

454             if (_zeroBytes(m)) return (false, new bytes(0));
```

Use assembly to emit events

Description:

Using the scratch space for event arguments (two words or fewer) will save gas over needing Solidity's full abi memory expansion used for emitting normally.

Severity:

G

Snippet:

```
File: contracts/TokenStaking.sol

61              emit RewardMultiplierUpdated(newMultiplier);
96              emit Staked(msg.sender, amount, duration);
112             emit Withdrawn(msg.sender, stakeIndex, info.amount);
128             emit ClaimedReward(msg.sender, stakeIndex, pending);
```

State variables only set in the constructor should be declared `immutable`

Description:

```
    Avoids a Gsset (**20000 gas**) in the constructor, and replaces the
first access in each transaction (Gcoldsload - **2100 gas**) and each
access thereafter (Gwarmacces - **100 gas**) with a `PUSH32` (**3 gas**).

    While `string`s are not value types, and therefore cannot be
`immutable`/`constant` if not hard-coded outside of the constructor, the
same behavior can be achieved by making the current contract `abstract`
with `virtual` functions for the `string` accessors, and having a child
contract override the functions with the hard-coded implementation-specific
values.
```

Severity:

G

Snippet:

```
File: contracts/DIVote.sol

33              voteToken = _voteToken;
```

Don't use `_msgSender()` if not supporting EIP-2771

Description:

```
    Use `msg.sender` if the code does not implement [EIP-2771 trusted
forwarder](https://eips.ethereum.org/EIPS/eip-2771) support
```

Severity:

G

Snippet:

```
File: @openzeppelin/contracts/utils/Context.sol

17          function _msgSender() internal view virtual returns (address) {
18              return msg.sender;
19          }
```

Use `uint256(1)` / `uint256(2)` instead for `true` and `false` boolean states

Description:

If you don't use boolean for storage you will avoid Gwarmaccess 100 gas. In addition, state changes of boolean from `true` to `false` can cost up to ~20000 gas rather than `uint256(2)` to `uint256(1)` that would cost significantly less.

Severity:

G

Snippet:

```
File: contracts/TokenStaking.sol

48          bool public stakingEnded;
```

Fewer storage slots can be used by storing timestamps in types smaller than uint256

Description:

Ethereum's block.timestamp can be stored in a type smaller than uint256. A uint32 variable can store a timestamp until year 2106.

Severity:

G

Snippet:

```
File: contracts/TokenStaking.sol

12              uint256 startTime;
13              uint256 endTime;
```

`++i` / `i++` should be `unchecked{++i}` / `unchecked{i++}` when it is not possible for them to overflow, as is the case when used in `for` - and `while` -loops

Description:

The `unchecked` keyword is new in solidity version 0.8.0, so this only applies to that version or higher, which these instances are. This saves 30-40 gas per loop.

Severity:

G

Snippet:

```
File: contracts/TokenStaking.sol

216             for (uint256 i = 0; i < stakes.length; ++i) {
```

`keccak256()` should only need to be called on a specific string literal once

Description:

It should be saved to an immutable variable, and the variable used instead. If the hash is being used as a part of a function selector, the cast to `bytes4` should also only be done once.

Severity:

G

Snippet:

```
File: @openzeppelin/contracts/utils/cryptography/EIP712.sol

38              keccak256("EIP712Domain(string name,string version,uint256
chainId,address verifyingContract)");
38              keccak256("EIP712Domain(string name,string version,uint256
chainId,address verifyingContract)");
```

Usage of `uints` / `ints` smaller than 32 bytes (256 bits) incurs overhead

Description:

> When using elements that are smaller than 32 bytes, your contract's gas usage may be higher. This is because the EVM operates on 32 bytes at a time. Therefore, if the element is smaller than that, the EVM must use more operations in order to reduce the size of the element from 32 bytes to the desired size.https://docs.soliditylang.org/en/v0.8.11/internals/layout*in*storage.htmlEach operation involving a `uint8` costs an extra **22-28 gas** (depending on whether the other operand is also a variable of type `uint8`) as compared to ones involving `uint256`, due to the compiler having to clear the higher bits of the memory word before operating on the `uint8`, as well as the associated stack operations of doing so. Use a larger size then downcast where needed.

Severity:

G

Snippet:

```
File: contracts/GasCreditVault.sol

424             (uint80 roundId, int256 price, , uint256 updatedAt, uint80
answeredInRound) = info.priceFeed.latestRoundData();
424             (uint80 roundId, int256 price, , uint256 updatedAt, uint80
answeredInRound) = info.priceFeed.latestRoundData();
454             (uint80 roundId, int256 price, , uint256 updatedAt, uint80
answeredInRound) = info.priceFeed.latestRoundData();
454             (uint80 roundId, int256 price, , uint256 updatedAt, uint80
answeredInRound) = info.priceFeed.latestRoundData();
462             uint8 priceFeedDecimals = info.priceFeed.decimals();
```

Consider activating `via-ir` for deploying

Description:

```
    The IR-based code generator was introduced with an aim to not only
allow code generation to be more transparent and auditable but also to
enable more powerful optimization passes that span across functions.

    You can enable it on the command line using `--via-ir` or with the
option `{"viaIR": true}`.

    This will take longer to compile, but you can just simple test it
before deploying and if you got a better benchmark then you can add --via-
ir to your deploy command

    More on: https://docs.soliditylang.org/en/v0.8.17/ir-breaking-
changes.html
```

Severity:

G

Snippet:

```
File: Various Files

None
```

Emit Used In Loop

Description:

Emitting an event inside a loop performs a LOG op N times, where N is the loop length. Consider refactoring the code to emit the event only once at the end of loop. Gas savings should be multiplied by the average loop length.

Severity:

G

Snippet:

```
File: contracts/GasCreditVault.sol
```

```
227                     emit EmergencyWithdrawn(token, balance);
401                     emit OwnerWithdrawn(token, tokenAmount,
deltaCredits);
```

Inverting the condition of an if-else-statement

Description:

Flipping the true and false blocks instead saves 3 gas.

Severity:

G

Snippet:

```
File: @openzeppelin/contracts/utils/structs/EnumerableSet.sol

69              if (!_contains(set, value)) {
70                  set._values.push(value);
71                  // The value is stored at length-1, but we add 1 to all
indexes
72                  // and use 0 as a sentinel value
73                  set._positions[value] = set._values.length;
74                  return true;
75              } else {
76                  return false;
77              }
```

`unchecked {}` can be used on the division of two `uints` in order to save gas

Description:

The division cannot overflow, since both the numerator and the denominator are non-negative.

Severity:

G

Snippet:

```
File: contracts/TokenStaking.sol

158                 uint256 reward = (duration * getDistributionRate()) / 1
days;
145             userStake.rewardDebt = (share * totalStakes.rewardIndex) /
1e18;
125             info.rewardDebt = (info.stakingShare *
totalStakes.rewardIndex) / 1e18;
181             return amount * getLockMultiplier(duration) *
getAmountMultiplier(amount) / 1e4;
192             if (duration == LockDuration.QUATERLY) return YEAR / 4;
163                 totalStakes.rewardIndex += (reward * 1e18) /
totalStakes.stakingShare;
176             return rewardMultiplier * DISTRIBITE_UNIT/ 1e18;
135             uint256 accumulated = (info.stakingShare *
totalStakes.rewardIndex) / 1e18;
193             if (duration == LockDuration.HALF_YEARLY) return YEAR / 2;
188             return getDistributionRate() * 365 * 1e18 /
totalStakes.stakingShare;
```

Private functions used once can be inlined

Description:

Private functions used once can be inlined to save GAS

Severity:

G

Snippet:

```
File: @openzeppelin/contracts/utils/ReentrancyGuard.sol

64          function _nonReentrantBefore() private {
65              // On the first call to nonReentrant, _status will be
NOT_ENTERED
66              if (_status == ENTERED) {
67                  revert ReentrancyGuardReentrantCall();
68              }
69
70              // Any calls to nonReentrant after this point will fail
71              _status = ENTERED;
72          }
74          function _nonReentrantAfter() private {
75              // By storing the original value once again, a refund is
```

```
triggered (see
76              // https://eips.ethereum.org/EIPS/eip-2200)
77              _status = NOT_ENTERED;
78          }
```

Low level call can be optimized with assembly

Description:

Low level call can be optimized with assembly. The returnData is copied to memory even if the variable is not utilized: the proper way to handle this is through a low level assembly call.

Severity:

G

Snippet:

```
File: contracts/MetaTxGateway.sol

231                     (bool refundSuccess, ) = payable(from).call{value:
refundAmount}("");
```

Use assembly to calculate hashes to save gas

Description:

Using assembly to calculate hashes can save 80 gas per instance

Severity:

G

Snippet:

```
File: contracts/TokenAirdrop.sol

45              bytes32 leaf = keccak256(abi.encodePacked(msg.sender,
amount));
```

Unused named return variables without optimizer waste gas

Description:

Consider changing the variable to be an unnamed one, since the variable is never assigned, nor is it returned by name. If the optimizer is not turned on, leaving the code as it is will also waste gas for the stack variable.

Severity:

G

Snippet:

```
File: contracts/MetaTxGateway.sol

261         ) internal view returns (bool valid) {
300         function _buildDomainSeparator() internal view returns (bytes32
domainSeparator) {
317         function calculateRequiredValue(MetaTransaction[] calldata
metaTxs) external pure returns (uint256 totalValue) {
326         function getNonce(address user) external view returns (uint256
currentNonce) {
335         function isRelayerAuthorized(address relayer) external view
returns (bool isAuthorized) {
343         function getDomainSeparator() external view returns (bytes32
separator) {
351         function getMetaTransactionTypehash() external pure returns
(bytes32 typehash) {
359         function getMainTypehash() external pure returns (bytes32
typehash) {
376         ) external view returns (bytes32 digest) {
411         function getTotalBatchCount() external view returns (uint256
count) {
419         function getVersion() external pure returns (string memory
version) {
```

Avoid contract existence checks by using low level calls

Description:

Prior to 0.8.10 the compiler inserted extra code, including `EXTCODESIZE` (100 gas), to check for contract existence for external function calls. In more recent

solidity versions, the compiler will not insert these checks if the external call has a return value. Similar behavior can be achieved in earlier versions by using low-level calls, since low level calls never check for contract existence.

Severity:

G

Snippet:

```
File: @openzeppelin/contracts-upgradeable/proxy/ERC1967/
ERC1967UpgradeUpgradeable.sol

160
AddressUpgradeable.functionDelegateCall(IBeaconUpgradeable(newBeacon).implement
data);
144
AddressUpgradeable.isContract(IBeaconUpgradeable(newBeacon).implementation()),
84                  try
IERC1822ProxiableUpgradeable(newImplementation).proxiableUUID() returns
(bytes32 slot) {
```

Avoid updating storage when the value hasn't changed

Description:

If the old value is equal to the new value, not re-storing the value will avoid a Gsreset (**2900 gas**), potentially at the expense of a Gcoldsload (**2100 gas**) or a Gwarmaccess (**100 gas**).

Severity:

G

Snippet:

```
File: contracts/TokenStaking.sol

59          function updateDistributeRate(uint256 newMultiplier) external
onlyOwner accrueReward {
64          function setStartTimeForStaking(uint256 startAt) external
onlyOwner {
```

Do not calculate constants

Description:

Due to how constant variables are implemented (replacements at compile-time), an expression assigned to a constant variable is recomputed each time that the variable is used, which wastes some gas.

Severity:

G

Snippet:

```
File: @openzeppelin/contracts/utils/Strings.sol

18          uint256 private constant SPECIAL_CHARS_LOOKUP =
19              (1 << 0x08) | // backspace
20                  (1 << 0x09) | // tab
21                  (1 << 0x0a) | // newline
22                  (1 << 0x0c) | // form feed
23                  (1 << 0x0d) | // carriage return
24                  (1 << 0x22) | // double quote
25                  (1 << 0x5c); // backslash
243         uint256 private constant ABS_MIN_INT256 = 2 ** 255;
```

Duplicated `require()`/`revert()` checks should be refactored to a modifier or function

Description:

Saves deployment costs.

Severity:

G

Snippet:

```
File: contracts/TeamAllocation.sol

55              require(index != NOT_FOUND, "No Member wallet");
```

The use of a logical AND in place of double if is slightly less gas efficient in instances where there isn't a corresponding else statement for the given if statement

Description:

Using a double if statement instead of logical AND (&&) can provide similar short-circuiting behavior whereas double if is slightly more efficient.

Severity:

G

Snippet:

```
File: contracts/TokenStaking.sol

156             if (block.timestamp > totalStakes.lastUpdatedAt &&
totalStakes.stakingShare > 0) {
```

Use the inputs/results of assignments rather than re-reading state variables

Description:

When a state variable is assigned, it saves gas to use the value being assigned, later in the function, rather than re-reading the state variable itself. If needed, it can also be stored to a local variable, and be used in that way. Both options avoid a Gwarmaccess (100 gas). Note that if the operation is, say +=, the assignment also results in a value which can be used. The instances below point to the first reference after the assignment, since later references are already covered by issues describing the caching of state variable values.

Severity:

G

Snippet:

```
File: contracts/TokenAirdrop.sol

46              require(MerkleProof.verify(proof, merkleRoot, leaf),
"Invalid proof");
```

Initializers can be marked payable

Description:

Payable functions cost less gas to execute, since the compiler does not have to add extra checks to ensure that a payment wasn't provided. An initializer can safely be marked as payable, since only the deployer would be able to pass funds, and the project itself would not pass any funds.

Severity:

G

Snippet:

```
File: contracts/MetaTxGateway.sol

87          function initialize() public initializer {
88              __Ownable_init();
89              __ReentrancyGuard_init();
90              __Pausable_init();
91              __UUPSUpgradeable_init();
92          }
```

Assigning state variables directly with named struct constructors wastes gas

Description:

Using named arguments for struct means that the compiler needs to organize the fields in memory before doing the assignment, which wastes gas. Set each field directly in storage (use dot-notation), or use the unnamed version of the constructor.

Severity:

G

Snippet:

```
File: contracts/TokenPresale.sol

65              paymentTokens[token] = PaymentToken({
```

Avoid fetching a low-level call's return data by using assembly

Description:

Even if you don't assign the call's second return value, it still gets copied to memory. Use assembly instead to prevent this and save 159 gas.

Severity:

G

Snippet:

```
File: contracts/MetaTxGateway.sol

165             (success, ) = target.call{value: value}(data);
231                     (bool refundSuccess, ) = payable(from).call{value:
refundAmount}("");
```

Using msg globals directly, rather than caching the value, saves gas

Description:

For example, use msg.sender directly rather than storing it to a local variable

Severity:

G

Snippet:

```
File: contracts/DIVote.sol

50              p.startTime = block.timestamp;
```

State variable read in a loop

Description:

The state variable should be cached in and read from a local variable, or accumulated in a local variable then written to storage once outside of the loop, rather than reading/updating it on every iteration of the loop, which will replace each Gwarmaccess (100 gas) with a much cheaper stack read.

Severity:

G

Snippet:

```
File: contracts/TeamAllocation.sol

42                  wallets.push(_wallets[i]);
95              for (uint256 i = 0; i < wallets.length; ++i) {
```

Storage re-read via storage pointer

Description:

The instances below point to the second+ access of a state variable, via a storage pointer, within a function. Caching the value replaces each Gwarmaccess (100 gas) with a much cheaper stack read.

Severity:

G

Snippet:

```
File: contracts/TokenStaking.sol

102             require(block.timestamp >= info.startTime +
getLockDuration(info.lockDuration) || stakingEnded, "Lock not expired");
```

State variables only set in their definitions should be declared constant

Description:

Avoids a Gsset (20000 gas) at deployment, and replaces the first access in each transaction (Gcoldsload - 2100 gas) and each access thereafter (Gwarmacces - 100 gas) with a PUSH32 (3 gas).

Severity:

G

Snippet:

```
File: @openzeppelin/contracts-upgradeable/proxy/utils/UUPSUpgradeable.sol

29          address private immutable __self = address(this);
```

State variables only set in the constructor should be declared immutable

Description:

Avoids a Gsset (20000 gas) in the constructor, and replaces the first access in each transaction (Gcoldsload - 2100 gas) and each access thereafter (Gwarmacces - 100 gas) with a PUSH32 (3 gas). While strings are not value types, and therefore cannot be immutable/constant if not hard-coded outside of the constructor, the same behavior can be achieved by making the current contract abstract with virtual functions for the string accessors, and having a child contract override the functions with the hard-coded implementation-specific values.

Severity:

G

Snippet:

```
File: contracts/DIVote.sol

24          ERC20Votes public voteToken;
```

Using this to access functions results in an external call, wasting gas

Description:

External calls have an overhead of 100 gas, which can be avoided by not referencing the function using this. Contracts are allowed to override their parents' functions and change the visibility from external to public, so make this change if it's required in order to call the function internally.

Severity:

G

Snippet:

```
File: contracts/MetaTxGateway.sol

148             try this._safeExecuteCall(metaTx.to, metaTx.value,
metaTx.data) returns (bool _success) {
```

Use local variables for emitting

Description:

Use the function/modifier's local copy of the state variable, rather than incurring an extra Gwarmaccess (100 gas). In the unlikely event that the state variable hasn't already been used by the function/modifier, consider whether it is really necessary to include it in the event, given the fact that it incurs a Gcoldsload (2100 gas), or whether it can be passed in to or back out of the functions that do use it

Severity:

G

Snippet:

```
File: contracts/DIVote.sol

58              emit ProposalCreated(proposalCount, _title, _quorum);
```

Using constants directly, rather than caching the value, saves gas

Description:

Severity:

G

Snippet:

```
File: @openzeppelin/contracts/utils/ReentrancyGuard.sol

71              _status = ENTERED;
```