ZK Governance Review

June 16, 2025

Prepared for ZKsync

Conducted by:

Kurt Willis (phaze)

Richie Humphrey (devtooligan)

About the ZKsync ZK Governance Review

ZKsync Era is a Layer 2 ZK rollup, a trustless protocol that uses cryptographic validity proofs to provide scalable and low-cost transactions on Ethereum. The `ZkMinterRateLimiterV1` is an extension of the `CappedMinter` contract that throttles the rate at which tokens can be streamed from the capped minter. This system provides controlled token distribution through rate-limited minting operations while maintaining administrative oversight through role-based access controls.

About Offbeat Security

Offbeat Security is a boutique security company providing unique security solutions for complex and novel crypto projects. Our mission is to elevate the blockchain security landscape through invention and collaboration.

Summary & Scope

The l2-contracts/src folder was reviewed at commit 6c3f74e.

The following **3 files** were in scope:

- l2-contracts/src/ZkMinterV1.sol
- l2-contracts/src/ZkMinterRateLimiterV1.sol
- l2-contracts/src/ZkMinterRateLimiterV1Factory.sol

The protocol implements rate-limited token minting through a windowed approach that tracks minting amounts over configurable time periods. The system uses role-based access control to manage minting permissions and administrative functions.

The review identified 1 MEDIUM, 2 LOW, and 2 INFORMATIONAL severity issues. Key findings include storage collision vulnerabilities in rate limiting logic and missing validation checks for critical parameters. Additional recommendations focus on improving the access control model and preventing deployment configuration errors.

FIX REVIEW:
After the initial review, we reviewed the fixes that were applied for our findings. These fixes are contained in commit 4890e.

REPO MIGRATION:
Subsequent to the fix review, the project migrated to a new repository. We confirmed that the files in scope in the new repository as of 3362a matched the files we reviewed in the fix review in the initial repo at commit 4890e.

Summary of Findings

Identifier	Title	Severity	Fixed
M-01	Rate limiter window size changes can cause storage collisions preventing legitimate minting	Medium	PR#63
L-01	Setting mint rate limit window to zero causes division by zero error	Low	PR#63
L-02	Missing address zero check in constructor	Low	PR#64
I-01	Redundant hashing in CREATE2 salt calculation	Informational	PR#66
I-02	Administrative functions remain callable after minter closure	Informational	PR#65

Detailed Findings

Medium Findings

[M-01] Rate limiter window size changes can cause storage collisions preventing legitimate minting

Summary

The ZkMinterRateLimiterV1 contract contains a vulnerability where changing the mint rate limit window size can cause storage collisions in the `mintedInWindow` mapping. This occurs because the window start calculation method changes while historical minting data persists, potentially preventing legitimate minting operations for extended periods.

Description

The current implementation uses a mathematical approach to calculate window start times based on the current `mintRateLimitWindow` size:

```
function currentMintWindowStart() public view returns (uint48) {
    return uint48(block.timestamp - ((block.timestamp - START_TIME) % mintRateLi
}
```

The contract tracks minted amounts using a mapping where the key is the calculated window start timestamp:

```
mapping(uint48 mintWindowStart => uint256 mintedAmount) public mintedInWindow;
```

When the window size is updated via `updateMintRateLimitWindow()`, the calculation method for `currentMintWindowStart()` changes, but historical data in the `mintedInWindow` mapping remains. This can cause the newly calculated window start timestamp to collide with a timestamp from a previous window configuration, leading the contract to incorrectly believe tokens have already been minted in the current window.

For example:

1. Start with a 1-day window and 100-token daily cap
2. Mint 100 tokens daily for 3 days (days 0, 1, 2)
3. On day 4, update the window to 7 days
4. The new 7-day window calculation may produce a start timestamp that matches one of the previous 1-day windows

5. The contract sees the historical 100 tokens as already minted in the current 7-day window
6. No additional minting is possible until the collision resolves, potentially blocking legitimate operations for days

Recommendation

Consider implementing the suggested approach using dedicated state variables instead of the mapping-based system:

```
uint48 public currentWindowStart;
uint256 public currentWindowMinted;

function mint(address _to, uint256 _amount) external {
    _revertIfClosed();
    _requireNotPaused();
    _checkRole(MINTER_ROLE, msg.sender);

    // Roll forward to new window if needed
    if (block.timestamp >= currentWindowStart + mintRateLimitWindow) {
        uint256 windowsPassed = (block.timestamp - currentWindowStart) / mintRate
        currentWindowStart += windowsPassed * mintRateLimitWindow;
        currentWindowMinted = 0;
    }

    // Check rate limit against current window
    if (currentWindowMinted + _amount > mintRateLimit) {
        revert ZkMinterRateLimiterV1__MintRateLimitExceeded(msg.sender, _amount)
    }

    currentWindowMinted += _amount;
    mintable.mint(_to, _amount);
    emit Minted(msg.sender, _to, _amount);
}

function updateMintRateLimitWindow(uint48 _mintRateLimitWindow) external {
    _checkRole(DEFAULT_ADMIN_ROLE, msg.sender);
    _updateMintRateLimitWindow(_mintRateLimitWindow);

    // Start fresh window when updating window size
    currentWindowStart = uint48(block.timestamp);
    currentWindowMinted = 0;
}
```

This approach eliminates storage collisions by maintaining explicit window state and provides more intuitive behavior when window sizes are updated. It also offers better gas efficiency by consistently writing to the same storage slots rather than creating new mapping entries.

Low Findings

[L-01] Setting mint rate limit window to zero causes division by zero error

Description

The `_updateMintRateLimitWindow()` function does not validate that the `_mintRateLimitWindow` parameter is greater than zero. This allows the admin to set `mintRateLimitWindow` to zero during contract initialization or through the `updateMintRateLimitWindow()` function. When `mintRateLimitWindow` is zero, calling `currentMintWindowStart()` results in a division by zero error due to the modulo operation:

```
function currentMintWindowStart() public view returns (uint48) {
    return uint48(block.timestamp - ((block.timestamp - START_TIME) % mintRateLi
}
```

Since `currentMintWindowStart()` is called within the `mint()` function, setting the rate limit window to zero effectively disables all minting functionality until the admin corrects the value.

Recommendation

Add validation with a custom error to ensure `_mintRateLimitWindow` is greater than zero:

```
error ZkMinterRateLimiterV1__InvalidRateLimitWindow();
function _updateMintRateLimitWindow(uint48 _mintRateLimitWindow) internal {
    if (_mintRateLimitWindow == 0) revert ZkMinterRateLimiterV1__InvalidRateLimi
    emit MintRateLimitWindowUpdated(mintRateLimitWindow, _mintRateLimitWindow);
    mintRateLimitWindow = _mintRateLimitWindow;
}
```

[L-02] Missing address zero check in constructor

Summary

The ZkMinterRateLimiterV1 contract allows deployment with a zero address admin, which permanently breaks the contract's core functionality. When deployed with `address(0)` as the admin, no one can grant the `MINTER_ROLE`, making token minting impossible and rendering the contract completely unusable.

Description

The constructor of ZkMinterRateLimiterV1 accepts an `_admin` parameter and grants both `DEFAULT_ADMIN_ROLE` and `PAUSER_ROLE` to this address without validating that it's not the zero address:

```
constructor(IMintable _mintable, address _admin, uint256 _mintRateLimit, uint48
    // ... other initialization code ...
```

```
    _grantRole(DEFAULT_ADMIN_ROLE, _admin);
    _grantRole(PAUSER_ROLE, _admin);
}
```

If `_admin` is set to `address(0)` during deployment, the contract becomes permanently non-functional because:

1. The `DEFAULT_ADMIN_ROLE` is granted to `address(0)`
2. Only accounts with `DEFAULT_ADMIN_ROLE` can grant the `MINTER_ROLE` to others
3. Since `address(0)` cannot execute transactions, no one can ever be granted the `MINTER_ROLE`
4. Without the `MINTER_ROLE`, the core `mint()` function becomes permanently inaccessible

This effectively creates a deployed contract that cannot fulfill its primary purpose of minting tokens.

Recommendation

Consider adding a zero address validation check in the constructor to prevent deployment with an invalid admin address:

```
constructor(IMintable _mintable, address _admin, uint256 _mintRateLimit, uint48 
+   require(_admin != address(0), "Admin cannot be zero address");
    _updateMintable(_mintable);
    _updateMintRateLimit(_mintRateLimit);
    _updateMintRateLimitWindow(_mintRateLimitWindow);
    START_TIME = uint48(block.timestamp);

    _grantRole(DEFAULT_ADMIN_ROLE, _admin);
    _grantRole(PAUSER_ROLE, _admin);
}
```

Informational Findings

[I-01] Redundant hashing in CREATE2 salt calculation

Description

The `_calculateSalt()` function in ZkMinterRateLimiterV1Factory includes constructor arguments in the salt calculation, even though these same arguments are already part of the CREATE2 address computation. This results in the same data being hashed twice, leading to unnecessary gas consumption without providing additional uniqueness benefits.

The current implementation hashes constructor arguments twice:

1. In the salt calculation: `keccak256(abi.encode(_args, block.chainid, _saltNonce))`

2. In the CREATE2 address calculation: `keccak256(abi.encode(_mintable, _admin, _mintRateLimit, _mintRateLimitWindow))`

Since CREATE2 addresses are computed using the formula `keccak256(0xff ++ deployer ++ salt ++ keccak256(bytecode + constructor_args))`, the constructor arguments already contribute to address uniqueness through the bytecode hash component.

Recommendation

Consider simplifying the salt calculation to avoid redundant hashing of constructor arguments:

```
function _calculateSalt(bytes memory _args, uint256 _saltNonce) internal view re
-    return keccak256(abi.encode(_args, block.chainid, _saltNonce));
+    return keccak256(abi.encode(block.chainid, _saltNonce));
}
```

[I-02] Administrative functions remain callable after minter closure

Description

The `ZkMinterV1` contract allows administrative functions like `updateMintable()` to be called even after the contract has been permanently closed. The `close()` function documentation states it "permanently closes the contract, preventing any future minting" and that "once closed, the contract cannot be reopened and all minting operations will be permanently blocked." However, the `updateMintable()` function does not check the closed state before execution.

While minting operations correctly call `_revertIfClosed()` to prevent execution when closed, administrative functions bypass this check:

```
function updateMintable(IMintable _mintable) external virtual {
    _checkRole(DEFAULT_ADMIN_ROLE, msg.sender);
    _updateMintable(_mintable); // No _revertIfClosed() call
}
```

This creates an inconsistency where the contract is considered "permanently closed" for minting purposes but remains mutable for administrative operations, potentially allowing unexpected state changes after closure.

Recommendation

Consider adding the closed state check to administrative functions to ensure complete immutability after closure:

```
function updateMintable(IMintable _mintable) external virtual {
+    _revertIfClosed();
     _checkRole(DEFAULT_ADMIN_ROLE, msg.sender);
     _updateMintable(_mintable);
}
```

Alternatively, if administrative functions should remain callable after closure, consider updating the documentation to clarify this behavior and explain the reasoning behind allowing these operations to continue.