// Security Assessment

11.25.2025 - 11.25.2025

zkPass Token Contract

ZKPass

HALBORN

Submit Feedback

zkPass Token Contract - ZKPass

Prepared by: HALBORN Last Updated 11/26/2025 Date of Engagement: November 25th, 2025 - November 25th, 2025

Summary

100% OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
7	0	0	0	0	7

1. INTRODUCTION

`ZKPass` engaged Halborn to conduct a security assessment on their smart contracts beginning and ending on November 25th, 2025. The security assessment was scoped to the smart contracts provided in the `zkPass-Token-Contract` GitHub repository, provided to the Halborn team. Commit hash and further details can be found in the Scope section of this report.

The reviewed contracts implement a fixed-supply omnichain ERC20 token that uses LayerZero OFT to burn on the source chain and mint on the destination chain during transfers. It also supports ERC20Permit for gasless approvals and ERC20Votes for governance.

TABLE OF CONTENTS

2. ASSESSMENT SUMMARY

`Halborn` was provided with 1 day for this engagement and assigned a full-time security engineer to assess the security of the smart contracts in scope. The assigned engineer possesses deep expertise in blockchain and smart contract security, including hands-on experience with multiple blockchain protocols.

The objective of this assessment is to:

- Identify potential security issues within the `ZKPToken` protocol smart contracts.
- Ensure that smart contract of `ZKPToken` protocol functions operate as intended.

In summary, `Halborn` identified several areas for improvement to reduce the likelihood and impact of risks, which were mostly addressed by the `ZKPass team`. The main ones were:

- `Implementing a mechanism to invalidate outstanding permit signatures prior to their expiry.`
- `Configuring the SUPPLY_CAP on non-mint chains to reflect the global supply and avoid ambiguity.`
- `Preventing bridging to address(0), as such transfers will reduce the actual circulating supply.`
- `Enforcing the global supply cap via the ERC20Votes supply guard.`
- `Preventing renouncement of ownership to avoid locking essential OFT configuration.`

3. TEST APPROACH AND METHODOLOGY

`Halborn` performed a combination of manual review of the code and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of the smart contract assessment. While manual testing is recommended to uncover flaws in logic, process, and implementation, automated testing techniques help enhance coverage of smart contracts and can quickly identify items that do not follow security best practices.

The following phases and associated tools were used throughout the term of the assessment:

- Research into the architecture and purpose of the `ZKPToken` protocol.
- Manual code review and walkthrough of the `ZKPToken` in-scope contracts.
- Manual assessment of critical `Solidity` variables and functions to identify potential vulnerability classes.
- Manual testing using custom scripts.
- Static Analysis of security for scoped contracts and imported functions. (`Slither`).
- Local deployment and testing with (`Foundry`, `Remix IDE`).

4. RISK METHODOLOGY

Every vulnerability and issue observed by Halborn is ranked based on **two sets** of **Metrics** and a **Severity Coefficient**. This system is inspired by the industry standard Common Vulnerability Scoring System.

The two **Metric sets** are: **Exploitability** and **Impact**. **Exploitability** captures the ease and technical means by which vulnerabilities can be exploited and **Impact** describes the consequences of a successful exploit.

The **Severity Coefficients** is designed to further refine the accuracy of the ranking with two factors: **Reversibility** and **Scope**. These capture the impact of the vulnerability on the environment as well as the number of users and smart contracts affected.

The final score is a value between 0-10 rounded up to 1 decimal place and 10 corresponding to the highest security risk. This provides an objective and accurate rating of the severity of security vulnerabilities in smart contracts.

The system is designed to assist in identifying and prioritizing vulnerabilities based on their level of risk to address the most critical issues in a timely manner.

4.1 EXPLOITABILITY

ATTACK ORIGIN (AO):

Captures whether the attack requires compromising a specific account.

ATTACK COST (AC):

Captures the cost of exploiting the vulnerability incurred by the attacker relative to sending a single transaction on the relevant blockchain. Includes but is not limited to financial and computational cost.

ATTACK COMPLEXITY (AX):

Describes the conditions beyond the attacker's control that must exist in order to exploit the vulnerability. Includes but is not limited to macro situation, available third-party liquidity and regulatory challenges.

METRICS:

EXPLOITABILITY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Origin (AO)	Arbitrary (AO:A) Specific (AO:S)	1 0.2
Attack Cost (AC)	Low (AC:L) Medium (AC:M) High (AC:H)	1 0.67 0.33
Attack Complexity (AX)	Low (AX:L) Medium (AX:M) High (AX:H)	1 0.67 0.33

Exploitability E is calculated using the following formula:

$$E = \prod m_e$$

4.2 IMPACT

CONFIDENTIALITY (C):

Measures the impact to the confidentiality of the information resources managed by the contract due to a successfully exploited vulnerability. Confidentiality refers to limiting access to authorized users only.

INTEGRITY (I):

Measures the impact to integrity of a successfully exploited vulnerability. Integrity refers to the trustworthiness and veracity of data stored and/or processed on-chain. Integrity impact directly affecting Deposit or Yield records is excluded.

AVAILABILITY (A):

Measures the impact to the availability of the impacted component resulting from a successfully exploited vulnerability. This metric refers to smart contract features and functionality, not state. Availability impact directly affecting Deposit or Yield is excluded.

DEPOSIT (D):

Measures the impact to the deposits made to the contract by either users or owners.

YIELD (Y):

Measures the impact to the yield generated by the contract for either users or owners.

METRICS:

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
Confidentiality (C)	None (C:N) Low (C:L) Medium (C:M) High (C:H) Critical (C:C)	0 0.25 0.5 0.75 1

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
Integrity (I)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1
Availability (A)	None (A:N) Low (A:L) Medium (A:M) High (A:H) Critical (A:C)	0 0.25 0.5 0.75 1
Deposit (D)	None (D:N) Low (D:L) Medium (D:M) High (D:H) Critical (D:C)	0 0.25 0.5 0.75 1
Yield (Y)	None (Y:N) Low (Y:L) Medium (Y:M) High (Y:H) Critical (Y:C)	0 0.25 0.5 0.75 1

Impact I is calculated using the following formula:

$$I = max(m_I) + \frac{\sum m_I - max(m_I)}{4}$$

4.3 SEVERITY COEFFICIENT

REVERSIBILITY (R):

Describes the share of the exploited vulnerability effects that can be reversed. For upgradeable contracts, assume the contract private key is available.

SCOPE (S):

Captures whether a vulnerability in one vulnerable contract impacts resources in other contracts.

METRICS:

SEVERITY COEFFICIENT (C)	COEFFICIENT VALUE	NUMERICAL VALUE
Reversibility (r)	None (R:N) Partial (R:P) Full (R:F)	1 0.5 0.25
Scope (s)	Changed (S:C) Unchanged (S:U)	1.25 1

Severity Coefficient C is obtained by the following product:

$$C = rs$$

The Vulnerability Severity Score S is obtained by:

$$S = min(10, EIC * 10)$$

The score is rounded up to 1 decimal places.

SEVERITY	SCORE VALUE RANGE
Critical	9 - 10
High	7 - 8.9
Medium	4.5 - 6.9
Low	2 - 4.4
Informational	0 - 1.9

5. SCOPE

REPOSITORY

(a) Repository: zkPass-Token-Contract

(b) Assessed Commit ID: 5abbef5

(c) Items in scope:

- src/Factory.sol
- src/ZKPToken.sol

Out-of-Scope: Third party dependencies and economic attacks.

REMEDIATION COMMIT ID:

- 7c89b3d

Out-of-Scope: New features/implementations after the remediation commit IDs.

6. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

7

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
NO MECHANISM TO INVALIDATE OUTSTANDING PERMIT SIGNATURES BEFORE EXPIRY	INFORMATIONAL	SOLVED - 11/25/2025
SUPPLY_CAP NOT REFLECTING GLOBAL SUPPLY ON NON-MINT CHAINS MAY LEAD TO INCORRECT ASSUMPTIONS	INFORMATIONAL	SOLVED - 11/25/2025
BRIDGING TO ADDRESS(0) MINTS TOKENS TO AN IRRECOVERABLE SINK, REDUCING ACTUAL CIRCULATING SUPPLY	INFORMATIONAL	SOLVED - 11/25/2025
GLOBAL SUPPLY CAP IS NOT ENFORCED BY ERC20VOTES SUPPLY GUARD	INFORMATIONAL	SOLVED - 11/25/2025
CHECKPOINT CLOCK MODE UNCLEAR AND MAY NOT MATCH INTENDED GOVERNANCE DESIGN	INFORMATIONAL	ACKNOWLEDGED - 11/26/2025
RENOUNCING OWNERSHIP CAN PERMANENTLY LOCK ESSENTIAL OFT CONFIGURATION	INFORMATIONAL	SOLVED - 11/25/2025

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
SINGLE-STEP OWNERSHIP TRANSFER INCREASES RISK OF MISCONFIGURED OR LOST OWNERSHIP	INFORMATIONAL	ACKNOWLEDGED - 11/26/2025

7. FINDINGS & TECH DETAILS

7.1 NO MECHANISM TO INVALIDATE OUTSTANDING PERMIT SIGNATURES BEFORE EXPIRY

// INFORMATIONAL

Description

The `ZKPToken` relies on `ERC20Permit` for `EIP-2612` permit functionality. Nonces for each user are incremented only when a permit call succeeds. There is no function that allows a user to proactively invalidate or skip their current nonce (for example, after signing a permit off-chain and later wanting to cancel it).

As a result, any previously signed but unused permit remains valid until its deadline elapses, as users have no on-chain mechanism to revoke such signatures early other than waiting for expiry or relying on the spender not to use them.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

Consider adding a user-controlled function that allows an account to increase its own permit nonce, invalidating any outstanding signatures that use lower nonces.

Remediation Comment

SOLVED: The **ZKPass team** solved the issue in the specified commit id by implementing a mechanism that increments the user nonce to invalidate their outstanding signatures.

Remediation Hash

https://github.com/zkPass-DAO/zkPass-Token-Contract/commit/7c89b3d0d5d65685dc000cf0bd06180d6011163e

7.2 SUPPLY_CAP NOT REFLECTING GLOBAL SUPPLY ON NON-MINT CHAINS MAY LEAD TO INCORRECT ASSUMPTIONS

// INFORMATIONAL

Description

The `ZKPToken` defines an immutable `SUPPLY_CAP` that is set only when `block.chainid == mintingChainId`. On the designated minting chain, `SUPPLY_CAP` is set to the fixed maximum supply and the entire amount is minted to the treasury. On all other chains, `SUPPLY_CAP` remains at its default value of zero.

Since the token design intends the initial mint on the minting chain to represent the global maximum supply across all chains, external integrators may reasonably expect `SUPPLY_CAP` on every chain to reflect that global cap. However, on non-mint chains the value is zero, which can be misinterpreted as no cap being configured on that chain or a local cap of zero contradicting the documented global 1 billion supply.

This does not break supply invariants, but it can cause confusion for other contracts that read `SUPPLY_CAP` on non-mint chains.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

The `SUPPLY_CAP` value should be initialized to the same fixed global constant within the constructor, regardless of the network on which the contract is deployed.

Remediation Comment

SOLVED: The **ZKPass team** solved the issue in the specified commit id by fixing the `SUPPLY_CAP` constant across all chains.

Remediation Hash

https://github.com/zkPass-DAO/zkPass-Token-Contract/commit/7c89b3d0d5d65685dc000cf0bd06180d6011163e

7.3 BRIDGING TO ADDRESS(0) MINTS TOKENS TO AN IRRECOVERABLE SINK, REDUCING ACTUAL CIRCULATING SUPPLY

// INFORMATIONAL

Description

When bridging via the `OFT.send()` function, the destination receiver is selected through `SendParam.to`. If `address(0)` is supplied (encoded as `bytes32`) either intentionally or accidentally, the OFT receive logic executes the following code, where in that case, the bridged amount is minted to `0xdead`, which functions as an unusable dead-sink address:

```
78  function _credit(
79      address _to,
80      uint256 _amountLD,
81      uint32 /*_srcEid*/
82  ) internal virtual override returns (uint256 amountReceivedLD) {
83      if (_to == address(0x0)) _to = address(0xdead); // _mint(...) does not support address
84      // @dev Default OFT mints on dst.
85      _mint(_to, _amountLD);
86      // @dev In the case of NON-default OFT, the _amountLD MIGHT not be == amountReceivedLD
87      return _amountLD;
88  }
```

Tokens minted to `0xdead` are rendered irrecoverable, causing a permanent reduction in circulating supply across all chains. The total supply remains equal to `SUPPLY_CAP`, but a portion of that supply becomes permanently inaccessible.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

Validate that the destination address is not zero before executing a cross-chain send.

Remediation Comment

SOLVED: The **ZKPass team** solved the issue in the specified commit id by preventing bridging when the receiver on the destination chain is `address(0)`.

Remediation Hash

https://github.com/zkPass-DAO/zkPass-Token-Contract/commit/7c89b3d0d5d65685dc000cf0bd06180d6011163e

7.4 GLOBAL SUPPLY CAP IS NOT ENFORCED BY ERC20VOTES SUPPLY GUARD

// INFORMATIONAL

Description

The `ZKPToken` contract does not override `ERC20Votes._maxSupply()`, so `ERC20Votes` does not enforce any hard supply cap. This has no impact now because there is no external minting function, and only one chain performs the initial mint. However, if a new chain deployment enables minting, there is no built-in safeguard to prevent exceeding the intended supply cap.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

Override `ERC20Votes._maxSupply()` so the token enforces the global maximum supply at the `ERC20Votes` level.

Remediation Comment

SOLVED: The **ZKPass team** solved the issue in the specified commit id by overriding the `ERC20._maxSupply()` to return the `SUPPLY_CAP`.

Remediation Hash

https://github.com/zkPass-DAO/zkPass-Token-Contract/commit/7c89b3d0d5d65685dc000cf0bd06180d6011163e

7.5 CHECKPOINT CLOCK MODE UNCLEAR AND MAY NOT MATCH INTENDED GOVERNANCE DESIGN

// INFORMATIONAL

Description

The `ZKPToken` contract inherits the default block-based clock from `ERC20Votes`. If the governance model is expected to rely on timestamps (`ERC-6372` mode), the current behavior may lead to mismatches between how voting weight is calculated on-chain and how governance timing is interpreted off-chain.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

Clarify whether governance should be block-based or timestamp-based. If timestamp-based behavior is intended, override `clock()` and `CLOCK_MODE()` accordingly.

Remediation Comment

ACKNOWLEDGED: The **ZKPass team** acknowledged the risk of this finding.

7.6 RENOUNCING OWNERSHIP CAN PERMANENTLY LOCK ESSENTIAL OFT CONFIGURATION

// INFORMATIONAL

Description

The `ZKPToken` contract inherits OpenZeppelin Ownable, which exposes `renounceOwnership()`. If the owner (or a compromised owner key) calls `renounceOwnership()`, the contract becomes ownerless. For an OFT-based token, the owner role is typically required to adjust LayerZero configuration (peers, options, ...), so losing the owner permanently removes the ability to manage cross-chain settings or react to emergencies, which can leave the system stuck in a misconfigured or partially broken state.

BVSS

AO:S/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

Consider disabling `renounceOwnership()` by overriding it to revert.

Remediation Comment

SOLVED: The **ZKPass team** solved the issue in the specified commit id by overriding the `renounceOwnership()` function to revert.

Remediation Hash

https://github.com/zkPass-DAO/zkPass-Token-Contract/commit/7c89b3d0d5d65685dc000cf0bd06180d6011163e

7.7 SINGLE-STEP OWNERSHIP TRANSFER INCREASES RISK OF MISCONFIGURED OR LOST OWNERSHIP

// INFORMATIONAL

Description

The `ZKPToken` contract relies on single-step ownership transfer from OpenZeppelin Ownable. In this pattern, calling `transferOwnership()` immediately sets the new owner. If the target address is mistyped, not yet deployed as a multisig, or otherwise incorrect, control over critical admin functions (such as OFT peer configuration and emergency actions) can be irreversibly lost in a single transaction.

BVSS

AO:S/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

Use a two-step ownership pattern (for example, Ownable2Step or a custom acceptOwnership flow) so that ownership transfer requires the new owner to explicitly accept.

Remediation Comment

ACKNOWLEDGED: The **ZKPass team** acknowledged the risk of this finding.
