TOBYWORLD

Smart Contract Security Assessment

VERSION 1.0

AUDIT DATES: December 27th to December 28th, 2025

AUDITED BY: vard | milamber

Contents

ᛉ Introduction

Algiz conducted an independent review of the smart-contract system. This engagement focused on evaluating the correctness of core protocol logic, identifying vulnerabilities that could lead to loss of funds or protocol manipulation, and assessing resilience under adversarial conditions.

This report summarizes the issues identified during the review, provides severity classifications, and offers actionable recommendations aimed at strengthening the protocol's security posture. The findings presented here reflect the codebase at the specified commit and may not apply to future revisions.

ᛟ About Algiz

Algiz is an independent smart contract security team specializing in helping early-stage protocols launch with confidence. Our team has identified 30+ Critical, High and Medium severity vulnerabilities across DeFi protocols, GameFi projects, and cross-chain systems through competitive audits and security research.

We combine engineering depth with product thinking - reviewing protocols not only for correctness, but for operational risk, upgrade safety, and long-term sustainability. Our founder-to-founder approach means we're a security partner, not just a vendor.

ᚾ Disclaimer

This report documents Algiz's observations based on a time-boxed review of the supplied codebase at the specified commit. The presented conclusions reflect only this snapshot of the system.

While every reasonable effort has been made to identify vulnerabilities, **no security review can guarantee** the complete absence of bugs, exploits, or unexpected behaviors. Smart-contract systems operate in adversarial, permissionless environments, and security must be treated as an ongoing process.

The mitigation recommendations included in this report are provided as guidance based on the information available during the engagement. Because this was a time-boxed review, complex or large-scale remediation efforts may require extended follow-up. In cases where fixes introduce substantial refactoring or modify core logic, we strongly advise conducting an additional full review to ensure the changes do not introduce new risks.

Subsequent security reviews, bug-bounty programs, and continuous on-chain monitoring are strongly recommended to maintain long-term protocol safety.

Risk Classification

Severity Level	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Informational

Impact	Description
High	Leads to a significant material loss of assets in the protocol, significantly harms a group of users, or disrupts a core functionality.
Medium	Leads to a moderate material loss of assets in the protocol, moderately harms a group of users, or affects ancillary functionalities.
Low	Leads to a minor material loss of assets in the protocol or to any unexpected behavior with some of the protocol's functionalities, but does not meet the criteria for higher severity.

Likelihood	Description
High	Attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
Medium	Only a conditionally incentivized attack vector, but still relatively likely. Vectors that require larger amount of capital to exercise relative to the amount gained or disruption of the protocol.
Low	Has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

ᛉ Executive Summary

Algiz conducted a focused security assessment of Tobyworld's *PATIENCE* and *TABOSHI* token contracts deployed on *Base*. The review identified 1 Low severity issue and 4 Informational observations, with no Critical, High, or Medium vulnerabilities discovered. The codebase demonstrates generally sound security practices. The issues identified are primarily related to code hygiene and standards compliance rather than exploitable vulnerabilities:

The Low severity finding ([L-1]) involves a misconfigured ERC-721 receiver hook that, while not affecting current functionality, would prevent the contract from safely receiving Uniswap V3 position NFTs in the future. Informational items include an ineffective deadline parameter in Uniswap interactions, a legacy tx.origin access control pattern in an already-renounced Ownable implementation, a missing reentrancy guard initializer, and a minor EIP-20 compliance deviation in the decimals() return type.

Overall Assessment: The reviewed *PATIENCE* and *TABOSHI* contracts present a solid security posture with no immediate risk to user funds. The Low severity finding should be addressed in any future contract iterations, and the informational items noted for ongoing code hygiene.

ᚨ About Protocol

Tobyworld is a community-driven DeFi ecosystem built on Base. The ecosystem operates as a multi-tiered, gamified economy centered around three core tokens:

- TOBY: The primary token and entry point into the ecosystem. Deployed as a standard ERC-20 token with no presale, VC allocations, or insider distributions.
- PATIENCE: A hyper-deflationary utility token designed as a "time-key" that unlocks higher yields on staking protocols and grants whitelist access for NFT mints. Holding PATIENCE is expected to enable governance participation in the Toby DAO.
- TABOSHI: Represents the highest tier of community commitment, functioning as yield multipliers within the staking system.

The project emphasizes on-chain transparency and decentralized governance, with the Toad Gang Telegram community serving as the primary coordination hub with more than 130,000 members.

ᛗ Scope and Methodology

This security assessment covers the following deployed token contracts on Base with a thorough manual review:

PATIENCE	ERC20	`0x6D96f18F00B815B2109A3766E79F6A7aD7785624`
TABOSHI	ERC20Z	`0x3A1a33cf4553Db61F0db2c1e1721CD480b02789f`

↑ Issues Found

Severity	**Total**	Acknowledged	Resolved
Critical Risk	0	0	0
High Risk	0	0	0
Medium Risk	0	0	0
Low Risk	1	0	0
Informational	4	0	0
Total Issues	**5**	**0**	**0**

ᛉ Findings

ID	Title	Severity	Status
L-1	`onERC721Received()` is never triggered on `activate()` and uses incorrect sender validation	Low	Reported
I-1	Ineffective use of `block.timestamp` for deadline	Informational	Reported
I-2	Use of `tx.origin` for access control is unsafe and enables phishing-style privilege escalation when an owner exists	Informational	Reported
I-3	Missing reentrancy guard initialization breaks OZ consistency	Informational	Reported
I-4	EIP20 Compliance - `PatienceToken:decimals()` returns `uint256` instead of `uint8`	Informational	Reported

ᚦ Critical

No critical vulnerabilities were identified during this assessment.

ᚺ High

No high severity vulnerabilities were identified during this assessment.

ᛗ Medium

No medium severity vulnerabilities were identified during this assessment.

ᛚ Low

The following Low severity vulnerabilities were identified during this assessment:

[L-1] onERC721Received() is never triggered on activate() and uses incorrect sender validation

IMPACT:	Low	**SEVERITY:**	Low
LIKELIHOOD:	Medium	**STATUS:**	Reported

Target
`ERC20z.sol`

Description
The `ERC20Z::onERC721Received()` hook appears intended to handle receipt of Uniswap V3 position NFTs. However, during position creation, `NonfungiblePositionManager.mint()` calls `_mint()`, not `_safeMint()`. Since `_mint()` does not invoke `onERC721Received()`, the hook is never executed as part of the normal liquidity-initialization flow and is effectively dead code.

In addition, the hook checks that `msg.sender` equals the Uniswap pool address. This condition is incorrect: for any ERC-721 `safeTransferFrom` / `_safeMint`, `msg.sender` will always be the ERC-721 contract itself (in this case, the `NonfungiblePositionManager`), not the underlying pool. As a result, if a Uniswap V3 position NFT is ever transferred to the ERC20Z contract using a *safe* transfer, the call will revert.

Impact

- Current core protocol behavior is unaffected.

  `onERC721Received()` is not triggered during Uniswap V3 position creation because `mint()` uses `_mint()` (not `_safeMint()`), so the incorrect logic does not break pool activation.

- However, the hook cannot safely receive Uniswap LP NFTs.

  If a Uniswap V3 position NFT is later sent to the ERC20Z contract using `safeTransferFrom` or `_safeMint`, the transfer will always revert because:

  - `msg.sender` will be the `NonfungiblePositionManager`, not the pool, and
  - the access-control check rejects the transfer.

Recommendation

The `msg.sender` should validate against the `NonfungiblePositionManager` contract address, not the pool:

```
 function onERC721Received(
    address from,
    address operator,
    uint256 tokenId,
    bytes memory data
 ) public virtual override returns (bytes4) {

-    if (msg.sender != _getERC20ZStorage().pool) {
+    if (msg.sender != address(nonfungiblePositionManager)) {
     revert OnlySupportReceivingERC721UniswapPoolNFTs();
    }

    return super.onERC721Received(from, operator, tokenId, data);
 }
```

Informational

The following informational observations were noted during this assessment:

[I-1] Ineffective use of block.timestamp for deadline

IMPACT:	Low	**SEVERITY:**	Informational
LIKELIHOOD:	Low	**STATUS:**	Reported

Target

`ERC20z.sol`

Description

In `ERC20Z:activate()`, the deadline for minting the initial position is set to block.timestamp:

```
INonfungiblePositionManager.MintParams memory params = INonfungiblePositionManager.MintParams({
        token0: token0,
        token1: token1,
        fee: UniswapV3LiquidityCalculator.FEE,
        tickLower: UniswapV3LiquidityCalculator.TICK_LOWER,
        tickUpper: UniswapV3LiquidityCalculator.TICK_UPPER,
        amount0Desired: amount0,
        amount1Desired: amount1,
        amount0Min: 0,
        amount1Min: 0,
        recipient: address(this),
        deadline: block.timestamp   //@audit [L] - deadline is set to the current timestamp
    });
```

Inside `NonfungiblePositionManager`, the deadline is checked as:

```
require(_blockTimestamp() <= params.deadline, "Transaction too old");
```

Within a single transaction, `block.timestamp` is constant. Since `activate()` and `mint()` are executed in the same transaction, this check effectively becomes:

```
require(block.timestamp <= block.timestamp, "Transaction too old");
```

which always passes. Because the deadline is computed at execution time, the call never expires and provides no protection against unintended late execution.

Impact

The time-based safety feature provided by Uniswap's `deadline` parameter is effectively disabled for the initial liquidity mint.

While this is not a direct security vulnerability (the pool is created and minted in a single transaction), it removes an otherwise standard safeguard against unintended late execution.

Recommendation

Allow the caller of `activate()` to specify a deadline and pass it through to the Uniswap mint call, instead of hardcoding `block.timestamp`

[I-2] Use of tx.origin for access control is unsafe and enables phishing-style privilege escalation when an owner exists

IMPACT:	Low	**SEVERITY:**	Informational
LIKELIHOOD:	Low	**STATUS:**	Reported

Target

`Ownable`

Description

The `PATIENCE` token contract inherits an `Ownable` implementation that relies on `tx.origin` for access control:

```
  modifier onlyOwner() {
    require(owner() == tx.origin, "Ownable: caller is not the owner");
    _;
  }
```

Using `tx.origin` for authorization is considered insecure. This is because `tx.origin` always refers to the **externally owned account (EOA)** that originally initiated the transaction - even if the call passes through one or more contracts.

This allows a malicious contract to trick the owner into calling it (for example, via a phishing UI). The malicious contract can then forward a call into `PATIENCE` token. In that forwarded call:

- `msg.sender` = the malicious contract
- `tx.origin` = the owner's EOA

Since the `onlyOwner` modifier checks `tx.origin`, the call will **incorrectly pass authorization**, granting the malicious contract owner-level privileges.

This weakens the security of any owner-restricted function in the contract hierarchy.

Impact

- Before `owner` is set to zero address by `renounceOwnership()`, a phishing interaction can allow an attacker-controlled contract to execute privileged actions and potentially transfer ownership, recover ERC20 or update fee parameters.
- This breaks the common security assumption that only the true owner account can trigger privileged actions.
- While the current `PatienceToken` setup does not rely on ownership (contract's `owner` is already `address(0)`, the inherited pattern is unsafe and could become exploitable in future extensions, upgrades, or forks.

Recommendation
Use `msg.sender` instead of `tx.origin`

[I-3] Missing reentrancy guard initialization breaks OZ consistency

IMPACT:	Low	**SEVERITY:**	Informational
LIKELIHOOD:	Low	**STATUS:**	Reported

Target
`ERC20z`

Description
In `ERC20Z:initialize()`, the `ReentrancyGuard` is never properly initialized, leaving its internal `_status` variable at the default value 0 instead of `NOT_ENTERED (1)`, which is rather a clarity / consistency matter as OZ recommends calling `__ReentrancyGuard_init()` when initializing the contract (or newer versions).

This does not render the `nonReentrant` guard useless, since after the very first use it will reset the status to `NOT_ENTERED`.

Recommendation
Add the `ReentrancyGuard_init()` in the initializer:

```
function initialize(address collection, uint256 tokenId, string calldata name, string calldata symbol) external
initializer returns (address) {
    __ERC20_init(name, symbol);
+   __ReentrancyGuard_init();
    ERC20ZStorage storage erc20zStorage = _getERC20ZStorage();
```

[I-4] EIP20 Compliance - PatienceToken:decimals() returns uint256 instead of uint8

IMPACT:	Low	**SEVERITY:**	Informational
LIKELIHOOD:	Low	**STATUS:**	Reported

Target

`IERC20Metadata`

Description

`decimals()` uses a non-standard return type (`uint256` vs the common `uint8` in ERC20 metadata). Since it returns `18`, most integrations will decode it fine; this is primarily a standards-consistency / tooling-compatibility note.