SantasList Audit Report

Prepared by Putfor

January 8, 2026

Table of Contents

Protocol Summary

The SantasList protocol consists of a primary ERC721 contract (`SantasList`) and an associated ERC20 token (`SantaToken`). The protocol assigns users one of several statuses (`NICE`, `EXTRA_NICE`, `NAUGHTY`, `UNKNOWN`) using a two-step verification process intended to be performed exclusively by a privileged Santa role. Based on the final status, eligible users can mint a Christmas NFT, while `EXTRA_NICE` users additionally receive SantaTokens.

SantaTokens are designed to be used to purchase NFTs for other users via the `buyPresent` function. The protocol relies on strict role separation and correct business logic enforcement to ensure fair NFT distribution and prevent unauthorized minting or token misuse.

Disclaimer

Putfor makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security research by Putfor is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the Solidity implementation of the contracts.

Risk Classification

		Impact		
		High	Medium	Low
	High	H	H/M	M
Likelihood	Medium	H/M	M	M/L
	Low	M	M/L	L

I use the CodeHawks `https://docs.codehawks.com/hawks-auditors/how-to-evaluate-a-finding-severity` severity matrix to determine severity. See the documentation for more details.

Audit Details

The findings described in this document correspond the following commit hash:

```
91c8f8c94a9ff2db91f0ab2b2742cf1739dd6374
```

Scope

```
./src/
#-- SantaToken.sol
#-- SantasList.sol
#-- TokenUri.sol
```

Compatibilities

- Solc Version: 0.8.22
- Chain(s) to deploy contract to:
  - Arbitrum
- Tokens
  - `SantaToken`

Roles

- `Santa` - Deployer of the protocol, should only be able to do 2 things:
  - `checkList` - Check the list once
  - `checkTwice` - Check the list twice
  - Additionally, it's OK if Santa mints themselves tokens.
- `User` - Can `buyPresents` and mint NFTs depending on their status of NICE, NAUGHTY, EXTRA-NICE or UNKNOWN

Executive Summary

This security assessment of the `SantasList` protocol was conducted using a methodology focused primarily on manual code review and targeted testing. The core analysis relied on in-depth inspection of the smart contracts to identify access control issues, business logic flaws, and deviations between the documented behavior and the actual implementation. Foundry was extensively used to write and execute custom tests demonstrating exploitability and validating identified issues.

Automated static analysis tools, including Slither and Aderyn, were also applied during the review process. While these tools did not surface any findings included in the final report, they served as supplementary checks alongside the manual analysis.

The audit identified a range of issues spanning multiple severity levels, including high-impact business logic vulnerabilities that allow unauthorized asset usage, medium-risk flaws affecting protocol fairness and trust assumptions, and low-risk issues related to access control enforcement, documentation mismatches, and code quality. Several findings demonstrate how users can exploit insufficient authorization checks and flawed validation logic to gain unintended advantages within the system.

Issues found

Severity	Number of issues found
High	3
Low	2
Total	5

Findings

High

[H-1] Unlimited NFT minting due to balance-based eligibility check

Description

The `collectPresent` function determines whether a user has already received their NFT by checking the current token balance of the caller:

```
    function collectPresent() external {
        if (block.timestamp < CHRISTMAS_2023_BLOCK_TIME) {
            revert SantasList__NotChristmasYet();
        }
@>      if (balanceOf(msg.sender) > 0) {
            revert SantasList__AlreadyCollected();
        }
        if (s_theListCheckedOnce[msg.sender] == Status.NICE &&
s_theListCheckedTwice[msg.sender] == Status.NICE) {
            _mintAndIncrement();
            return;
        } else if (
```

```
            s_theListCheckedOnce[msg.sender] == Status.EXTRA_NICE
                && s_theListCheckedTwice[msg.sender] == Status.EXTRA_NICE
        ) {
            _mintAndIncrement();
            i_santaToken.mint(msg.sender);
            return;
        }
        revert SantasList__NotNice();
    }
```

This approach assumes that once a user receives an NFT, they will always hold it. However, NFTs are transferable by design. After successfully minting an NFT, a user can simply transfer it to another address. Once the balance becomes zero again, the same address can call `collectPresent` again and pass the balance check.

Because there is no additional state tracking to mark that an address has already claimed its present, this process can be repeated indefinitely, allowing the same user to mint an unlimited number of NFTs.

Risk

Likelihood: High

Exploiting this issue requires no special permissions, no timing assumptions, and no complex setup. Any user with basic knowledge of NFT transfers can repeatedly call `collectPresent` after transferring the minted NFT away. The attack is trivial to execute and highly likely to be abused once discovered.

Impact: High

An attacker can mint an unlimited number of NFTs, which can severely dilute the collection, break scarcity assumptions, and undermine the economic and reputational value of the project. If the NFTs have secondary market value or utility, this issue can directly lead to financial losses and loss of trust in the protocol.

Proof of Concept

To transfer an NFT, its `tokenId` must be known. In this protocol, token IDs are generated sequentially using the following state variable:

```
uint256 private s_tokenCounter;
```

Since this variable has `private` visibility, it cannot be accessed directly from the contract interface. However, it can still be read from contract storage by identifying the storage slot in which it is stored.

To determine the storage slot, the following Foundry command can be used:

```
forge inspect SantasList storage
```

This command outputs the contract's storage layout. From the output, we can observe that `s_tokenCounter` is stored in slot `8`:

```
┌------------------+----------+------+--------+-------+----------------┐
| Name             | Type     | Slot | Offset | Bytes | Contract       |
|------------------+----------+------+--------+-------+----------------|
| s_tokenCounter   | uint256  | 8    | 0      | 32    | SantasList.sol |
└------------------+----------+------+--------+-------+----------------┘
```

To demonstrate the exploit, an additional address is introduced in the test file to receive transferred NFTs:

```
contract SantasListTest is Test {
    SantasList santasList;
    SantaToken santaToken;

    address user = makeAddr("user");
    address santa = makeAddr("santa");
+   address secondAddress = makeAddr("secondAddress");
    _CheatCodes cheatCodes = _CheatCodes(VM_ADDRESS);
```

Below is the full Proof of Concept test.

First, a helper function is used to read the current token ID by loading the value of `s_tokenCounter` directly from storage. Since the counter is incremented after minting, the actual token ID of the last minted NFT is `s_tokenCounter - 1`.

```
function getTokenId() public view returns (uint256) {
    // Load the raw value of s_tokenCounter from storage slot 8
    bytes32 rawTokenId = vm.load(address(santasList), bytes32(uint256(8)));

    // Convert the loaded value to uint256
    uint256 trueTokenId = uint256(rawTokenId);

    // The last minted tokenId is counter - 1
    return trueTokenId - 1;
}
```

The main test demonstrates how a user can repeatedly mint NFTs by transferring them away between calls to `collectPresent`.

```
function testMintExtraNft() public {
    // Santa marks the user as EXTRA_NICE in both checks
    vm.startPrank(santa);
    santasList.checkList(address(user), SantasList.Status.EXTRA_NICE);
    santasList.checkTwice(address(user), SantasList.Status.EXTRA_NICE);
    vm.stopPrank();

    // Move time forward to after Christmas
    vm.warp(santasList.CHRISTMAS_2023_BLOCK_TIME() + 1);

    vm.startPrank(user);

    // First mint
    santasList.collectPresent();
    // Transfer the minted NFT away, resetting the user's balance to zero
    santasList.transferFrom(user, secondAddress, getTokenId());
```

```
    // Second mint and transfer
    santasList.collectPresent();
    santasList.transferFrom(user, secondAddress, getTokenId());

    // Third mint and transfer
    santasList.collectPresent();
    santasList.transferFrom(user, secondAddress, getTokenId());

    vm.stopPrank();

    // The receiving address ends up holding three NFTs
    assertEq(santasList.balanceOf(secondAddress), 3);
}
```

The test can be executed using:

```
forge test --match-test testMintExtraNft
```

Output:

```
Ran 1 test for test/unit/SantasListTest.t.sol:SantasListTest
[PASS] testMintExtraNft() (gas: 286284)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 9.48ms (408.13µs
CPU time)

Ran 1 test suite in 86.00ms (9.48ms CPU time): 1 tests passed, 0 failed, 0
skipped (1 total tests)
```

As demonstrated by the test above, a user can mint multiple NFTs by repeatedly transferring the previously minted NFT to another address. This confirms that the balance-based eligibility check can be bypassed, allowing an attacker to mint an unlimited number of NFTs.

Recommended Mitigation

Introduce a dedicated mapping that records whether a user has already claimed a present:

```
/*//////////////////////////////////////////////////////////////
                            STATE VARIABLES
    //////////////////////////////////////////////////////////////*/
    mapping(address person => Status naughtyOrNice) private
s_theListCheckedOnce;
    mapping(address person => Status naughtyOrNice) private
s_theListCheckedTwice;
+   mapping(address person => bool isCollected) private s_hasCollected;
    address private immutable i_santa;
    uint256 private s_tokenCounter;
    SantaToken private immutable i_santaToken;
```

This mapping should be used as the single source of truth for determining eligibility to collect a present.

```
function collectPresent() external {
    if (block.timestamp < CHRISTMAS_2023_BLOCK_TIME) {
        revert SantasList__NotChristmasYet();
```

```
    }
-   if (balanceOf(msg.sender) > 0) {
+   if (s_hasCollected[msg.sender]) {
        revert SantasList__AlreadyCollected();
    }

    if (
        s_theListCheckedOnce[msg.sender] == Status.NICE
        && s_theListCheckedTwice[msg.sender] == Status.NICE
    ) {
+       s_hasCollected[msg.sender] = true;
        _mintAndIncrement();
        return;
    }
    else if (
        s_theListCheckedOnce[msg.sender] == Status.EXTRA_NICE
        && s_theListCheckedTwice[msg.sender] == Status.EXTRA_NICE
    ) {
+       s_hasCollected[msg.sender] = true;
        _mintAndIncrement();
        i_santaToken.mint(msg.sender);
        return;
    }

    revert SantasList__NotNice();
}
```

[H-2] Unauthorized Burning of SantaTokens Allows Minting NFTs at Another User's Expense

Description

The `buyPresent()` function is intended to allow users to purchase an NFT for someone else by spending SantaTokens. According to the protocol documentation, SantaTokens are earned only by users marked as `EXTRA_NICE` and are meant to be voluntarily spent by their holders.

```
function buyPresent(address presentReceiver) external {
   i_santaToken.burn(presentReceiver);
   _mintAndIncrement();
}
```

However, the current implementation of `buyPresent()` does not enforce that the caller is the owner of the SantaTokens being burned. Instead, it blindly calls `i_santaToken.burn(presentReceiver)`, using the provided `presentReceiver` address as the token source, while minting the NFT to the caller.

This allows a malicious user to specify the address of an `EXTRA_NICE` user who owns SantaTokens as `presentReceiver`. As a result, SantaTokens are burned from the victim's balance without their consent, and the attacker receives the newly minted NFT.

In practice, this means that any user — including those marked as `NAUGHTY` or `UNKNOWN` — can mint NFTs for themselves by forcibly spending SantaTokens belonging to other users.

Risk

Likelihood: High

- Exploitation is trivial and requires no special privileges
- The vulnerable function `buyPresent` is publicly accessible
- Attack does not depend on timing, randomness, or complex conditions
- Any address can target a SantaToken holder to mint NFTs at their expense

Impact: Medium

- Users can lose their SantaTokens without approving or initiating the action
- NFTs can be minted by unauthorized users using another user's tokens

---

Proof of Concept

To demonstrate this issue, an additional address is required to represent a malicious user with the `NAUGHTY` status. The following address is added to the existing test setup:

```
contract SantasListTest is Test {
    SantasList santasList;
    SantaToken santaToken;

    address user = makeAddr("user");
    address santa = makeAddr("santa");
+   address naughty = makeAddr("naughty");
    _CheatCodes cheatCodes = _CheatCodes(VM_ADDRESS);
```

Proof of Code

```
function testBuyPresentByNaughty() public {
    // Santa assigns statuses to users
    vm.startPrank(santa);
    santasList.checkList(user, SantasList.Status.EXTRA_NICE);
    santasList.checkTwice(user, SantasList.Status.EXTRA_NICE);
    santasList.checkList(naughty, SantasList.Status.NAUGHTY);
    santasList.checkTwice(naughty, SantasList.Status.NAUGHTY);
    vm.stopPrank();

    // Move time forward to allow present collection
    vm.warp(santasList.CHRISTMAS_2023_BLOCK_TIME() + 1);

    // EXTRA_NICE user collects their present and receives SantaTokens
    vm.prank(user);
    santasList.collectPresent();

    // NAUGHTY user abuses buyPresent by burning SantaTokens from the victim
    vm.prank(naughty);
    santasList.buyPresent(user);

    // The victim keeps their original NFT but loses SantaTokens
    assertEq(santasList.balanceOf(user), 1);
    assertEq(santaToken.balanceOf(user), 0);

    // The attacker receives an NFT without owning SantaTokens
    assertEq(santasList.balanceOf(naughty), 1);
```

To reproduce the issue, run the following command:

```
forge test --match-test testBuyPresentByNaughty
```

Output:

```
Ran 1 test for test/unit/SantasListTest.t.sol:SantasListTest
[PASS] testBuyPresentByNaughty() (gas: 263345)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 10.84ms (183.04µs
CPU time)

Ran 1 test suite in 93.02ms (10.84ms CPU time): 1 tests passed, 0 failed, 0
skipped (1 total tests)
```

As shown by the test, a user with the `NAUGHTY` status can mint an NFT for themselves by forcibly burning SantaTokens belonging to another user. This confirms the presence of a vulnerability that allows attackers to mint NFTs at the expense of other users without their consent.

Recommended Mitigation

Require that SantaTokens are burned from `msg.sender`, while allowing the NFT to be minted to the intended recipient.

```
+function _mintAndIncrementTo(address recipient) private {
+    _safeMint(recipient, s_tokenCounter++);
+}

function buyPresent(address presentReceiver) external {
-    i_santaToken.burn(presentReceiver);
-    _mintAndIncrement();
+    i_santaToken.burn(msg.sender);
+   _mintAndIncrementTo(presentReceiver);
}
```

[H-3] The `checkList` function lacks access control, allowing anyone to modify user status

---

Description

The `checkList` function is intended to mark a person as **NICE**, **EXTRA_NICE**, or **NAUGHTY** during the first verification step, and according to the documentation, only the **Santa** account should be able to call it.

```
  /*
   * @notice Do a first pass on someone if they are naughty or nice.
@> * Only callable by santa
   *
   * @param person The person to check
   * @param status The status of the person
   */
@> function checkList(address person, Status status) external {
        s_theListCheckedOnce[person] = status;
        emit CheckedOnce(person, status);
    }
```

However, the function currently has no modifier and performs no access control checks. Any address can call it and update the `s_theListCheckedOnce` mapping, altering a user's first-step status arbitrarily.

Because there is no restriction, this can be done without permissions or special conditions.

Risk

Likelihood: High

- The function is publicly accessible
- Exploitation requires no privileges or setup
- Any user can call it to modify first-step status for themselves or others
- No timing, randomness, or complex conditions are required

Impact: High

- Unauthorized users can modify the first-step status of any address, violating the protocol's intended access control model
- This weakens the integrity and trust assumptions of the list verification process

---

Proof of Concept

To demonstrate the vulnerability, we create a simple test showing that any user can call the `checkList` function and modify their first-step status without any restrictions.

```
function testUserCanCallCheckList() public {
    // Impersonate a regular user
    vm.prank(user);

    // Call checkList to set the user's status to EXTRA_NICE
    santasList.checkList(user, SantasList.Status.EXTRA_NICE);

    // Verify that the status was updated
    assert(santasList.getNaughtyOrNiceOnce(user) ==
SantasList.Status.EXTRA_NICE);
}
```

The test can be executed using the following command:

```
forge test --match-test testUserCanCallCheckList
```

Output:

```
Ran 1 test for test/unit/SantasListTest.t.sol:SantasListTest
[PASS] testUserCanCallCheckList() (gas: 37575)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 1.83ms (101.84µs
CPU time)

Ran 1 test suite in 7.16ms (1.83ms CPU time): 1 tests passed, 0 failed, 0
skipped (1 total tests)
```

As demonstrated by the test, a user who is not Santa was able to set their desired status. This confirms the presence of an **access control vulnerability** in the `checkList` function.

Recommended Mitigation

Add an access control modifier to restrict calls to the `checkList` function exclusively to the `santa` account.

```
- function checkList(address person, Status status) external {
+ function checkList(address person, Status status) external onlySanta {
        s_theListCheckedOnce[person] = status;
        emit CheckedOnce(person, status);
}
```

Low

[L-1] Unused Constant Variable: `PURCHASED_PRESENT_COST`

Description

The contract declares a constant `PURCHASED_PRESENT_COST` intended to represent the cost of SantaTokens required for naughty users to buy presents. However, this variable is never referenced or used anywhere else in the codebase.

```
    // This variable is ok even if it's off by 24 hours.
    uint256 public constant CHRISTMAS_2023_BLOCK_TIME = 1_703_480_381;
@>  // The cost of santa tokens for naughty people to buy presents
@>  uint256 public constant PURCHASED_PRESENT_COST = 2e18;
```

Impact

Low — adds unnecessary noise and slightly reduces code clarity.

Recommended Mitigation

Remove the unused constant `PURCHASED_PRESENT_COST` from the contract to improve code clarity and maintainability.

```
    // This variable is ok even if it's off by 24 hours.
    uint256 public constant CHRISTMAS_2023_BLOCK_TIME = 1_703_480_381;
-   // The cost of santa tokens for naughty people to buy presents
-   uint256 public constant PURCHASED_PRESENT_COST = 2e18;
```

[L-2] Missing Event on Gift Collection

Description

The `collectPresent()` function mints an NFT for users but does not emit any event indicating that the NFT was successfully minted or collected. Emitting an event would allow off-chain tools, analytics, and front-end applications to easily track gift collection and improve transparency.

Impact

Low — does not affect functionality, but reduces transparency and makes off-chain monitoring harder.

Recommended Mitigation

Emit an event, e.g., `PresentCollected(address user, uint256 tokenId)`, after successfully minting the NFT in `collectPresent()`.

```
    event CheckedOnce(address person, Status status);
    event CheckedTwice(address person, Status status);
+   event PresentCollected(address person, uint256 tokenId);
```

```
    function collectPresent() external {
        if (block.timestamp < CHRISTMAS_2023_BLOCK_TIME) {
            revert SantasList__NotChristmasYet();
        }
        if (balanceOf(msg.sender) > 0) {
            revert SantasList__AlreadyCollected();
        }
        if (s_theListCheckedOnce[msg.sender] == Status.NICE &&
s_theListCheckedTwice[msg.sender] == Status.NICE) {
            _mintAndIncrement();
+           emit PresentCollected(msg.sender, s_tokenCounter);
            return;
        } else if (
            s_theListCheckedOnce[msg.sender] == Status.EXTRA_NICE
                && s_theListCheckedTwice[msg.sender] == Status.EXTRA_NICE
        ) {
            _mintAndIncrement();
+           emit PresentCollected(msg.sender, s_tokenCounter);
            i_santaToken.mint(msg.sender);
            return;
        }
        revert SantasList__NotNice();
    }
```