our shielding · Your smart contracts, our shielding · Your smart c

shieldify

Spellborne

Season 2 Airdrop

SECURITY REVIEW

Date: 6 November 2025

CONTENTS

1. About Shieldify

Positioned as the first hybrid Web3 Security company, Shieldify shakes things up with a unique subscription-based auditing model that entitles the customer to unlimited audits within its duration, as well as top-notch service quality thanks to a disruptive 6-layered security approach. The company works with very well-established researchers in the space and have secured multiple millions in TVL across protocols, also can audit codebases written in Solidity, Vyper, Rust, Cairo, Move and Go.

Learn more about us at shieldify.org.

2. Disclaimer

This security review does not guarantee bulletproof protection against a hack or exploit. Smart contracts are a novel technological feat with many known and unknown risks. The protocol, which this report is intended for, indemnifies Shieldify Security against any responsibility for any misbehavior, bugs, or exploits affecting the audited code during any part of the project's life cycle. It is also pivotal to acknowledge that modifications made to the audited code, including fixes for the issues described in this report, may introduce new problems and necessitate additional auditing.

3. About Spellborne - Season 2 Airdrop

Overview

Airdrop contract for Season 2 with linear vesting and early unlock functionality for in-app purchases.

Requirements

Distribution Parameters

- **Total Allocation**: 5% of total token supply
- **Expected Recipients**: 6,000-7,000 wallets
- **Vesting Period**: 6 months linear vesting

Core Functionality

1. Linear Vesting

- Tokens unlock linearly over 6 months from the airdrop start date
- Users can claim unlocked tokens at any time
- Track claimed vs. claimable amounts per wallet

2. Locked Token Spending

Users can spend locked (unvested) tokens for in-app purchases before vesting completes.

Flow:

1. User creates off-chain payment request in app for amount X
2. Backend validates the request and calls the contract function
3. Contract transfers X locked tokens from the user's allocation to the treasury/backend wallet
4. Reduce the user's total vested allocation by X

Key Considerations:

- Must validate backend caller (only authorized backend addresses)
- Spending reduces both locked balance AND total allocation (not just advancing unlock)
- Cannot spend more than the remaining locked balance
- Already unlocked/claimed tokens remain unaffected

4. Risk Classification

Severity	**Impact: High**	**Impact: Medium**	**Impact: Low**
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

4.1 Impact

- **High** - results in a significant risk for the protocol's overall well-being. Affects all or most users
- **Medium** - results in a non-critical risk for the protocol affects all or only a subset of users, but is still unacceptable
- **Low** - losses will be limited but bearable - and covers vectors similar to griefing attacks that can be easily repaired

4.2 Likelihood

- **High** - almost certain to happen and highly lucrative for execution by malicious actors
- **Medium** - still relatively likely, although only conditionally possible
- **Low** - requires a unique set of circumstances and poses non-lucrative cost-of-execution to rewards ratio for the actor

5. Security Review Summary

The security review lasted 2 days, with a total of 32 hours dedicated to the audit by 2 researchers from the Shieldify team.

Overall, the code is well-written. The audit report contributed by identifying one Medium and three Low severity issues. They're mainly related to unfair vesting extension for users spending locked tokens before claiming and some other minor vesting misconfigurations.

The Spellborne team has done a great job with their test suite and provided support and responses to all of the questions that the Shieldify researchers had.

5.1 Protocol Summary

Project Name	**Spellborne - Season 2 Airdrop**
Repository	s2-airdrop
Type of Project	ERC721C NFT with Royalty Enforcement, Token Vesting
Security Review Timeline	2 days
Review Commit Hash	fa331efe0bc68794537d6df645241f61be14be7b
Fixes Review Commit Hash	f4b71f4157cccc78a1679e3a25a6612b2c729d9e

5.2 Scope

The following smart contracts were in the scope of the security review:

File	**nSLOC**
src/S2Airdrop.sol	153
Total	**153**

6. Findings Summary

The following number of issues have been identified, sorted by their severity:

- **Medium** issues: **1**
- **Low** issues: **3**
- **Info** issues: **3**

ID	**Title**	**Severity**	**Status**
[M-01]	Unfair Vesting Extension for Users Spending Locked Tokens Before Claiming	Medium	Fixed
[L-01]	Vesting Parameters Can Be Changed After Vesting Starts	Low	Acknowledged
[L-02]	Users Can Bypass Vesting During Initialization	Low	Acknowledged
[L-03]	Merkle Root Is Mutable Mid-Airdrop	Low	Acknowledged
[I-01]	Hardcoded Token Address Limits Deployment To A Specific Chain	Info	Acknowledged
[I-02]	No Recovery Path for non-`TOKEN` ERC-20s Sent to the Contract	Info	Acknowledged
[I-03]	Missing Dedicated Event for Owner Withdrawals (Monitoring Blind Spot)	Info	Acknowledged

7. Findings

[M-01] Unfair Vesting Extension for Users Spending Locked Tokens Before Claiming

Severity

Medium Risk

Description

The vesting calculation in `_vestedAmount()` creates an inequitable outcome for users who spend locked tokens before claiming vested tokens compared to users who claim first. The function reduces the total allocation by `spendLockedAmount` before applying the vesting percentage, effectively extending the vesting period for users who utilize the in-app purchase feature early.

When a user spends locked tokens via `spendLockTokens()`, the `spendLockedAmount` is incremented. Subsequently, `_vestedAmount()` calculates the vested amount as:

```
uint256 leftVestedAmount = details.vestedAmount - details.
    spendLockedAmount;
return (leftVestedAmount * elapsed) / duration;
```

This means the vesting percentage is applied to the reduced allocation rather than the original amount, penalizing users who spend locked tokens before the vesting period completes.

Location of Affected Code

File: src/S2Airdrop.sol#L285-L301

```
function _vestedAmount(address user) internal view returns (uint256) {
    Airdrop memory details = airdropDetails[user];

    if (block.timestamp < startTime) return 0;

    uint256 leftVestedAmount = details.vestedAmount - details.
        spendLockedAmount;

    if (block.timestamp >= endTime) return leftVestedAmount;

    uint256 elapsed = block.timestamp - startTime;
    uint256 duration = endTime - startTime;

    return (leftVestedAmount * elapsed) / duration;
}
```

Impact

Users who spend locked tokens before claiming face a significant disadvantage: their claimable vested amount is calculated on a reduced base, resulting in fewer tokens available for withdrawal at any given point during vesting. Conversely, users who claim vested tokens first can then spend the same amount of locked tokens while retaining their full vested allocation.

This creates an unfair system where identical users with identical allocations receive different treatment based solely on the order of their interactions with the contract.

Proof of Concept

Both User A and User B have 10,000 tokens allocated, with 3 months of vesting passed (50% vested).

User A (claims first): 1. Claims 5,000 vested tokens (50% of 10,000) - `details.claimedAmount = 5,000`

2. Spends 5,000 locked tokens via in-app purchase - `details.spendLockedAmount = 5,000` - Successfully completes the purchase

User B (spends first): 1. Spends 5,000 locked tokens via in-app purchase
- `details.spendLockedAmount = 5,000` 2. Attempts to claim 5,000 vested tokens (50% of original 10,000)
- `leftVestedAmount = 10,000 - 5,000 = 5,000` - \codex{_vestedAmount = 5,000 * 50% = 2,500}
- `availableAmount = 2,500 - 0 = 2,500` - Can only claim 2,500 tokens instead of the expected 5,000

User B receives 2,500 fewer claimable tokens for the same allocation and timeline simply because they utilized the in-app purchase feature before claiming.

Recommendation

Calculate the vested amount based on the original allocation first, then subtract spent locked tokens. Alternatively, if the vesting schedule extension is intentional, automatically claim the user's vested tokens before purchase calculations.

Team Response

Fixed.

[L-01] Vesting Parameters Can Be Changed After Vesting Starts

Severity

Low Risk

Description

The `setTime()` function only validates that the new `_startTime` is in the future, but does not check whether vesting has already commenced with the current stored `startTime`. This allows the owner to modify vesting parameters after users have already begun claiming tokens, potentially disrupting user expectations and previously calculated vesting schedules.

The validation in the function checks:

```
if (block.timestamp > _startTime) revert VestingAlreadyStarted();
```

However, this only prevents setting a `_startTime` that is already in the past. If the current `startTime` has already passed and vesting is active, the owner can still call `setTime()` with a future timestamp, effectively restarting or extending the vesting period.

Location of Affected Code

File: src/S2Airdrop.sol#L166-L174

```
function setTime(uint40 _startTime, uint40 _endTime) public onlyOwner {
    if (block.timestamp > _startTime) revert VestingAlreadyStarted();
    if (_startTime >= _endTime) revert VestingEndTimeError();

    startTime = _startTime;
    endTime = _endTime;

    emit VestingTimeSet(_startTime, _endTime);
}
```

Impact

The owner can change vesting parameters after vesting has started, leading to:

1. Users who have calculated their vesting schedule based on the original parameters may find their claimable amounts suddenly reduced or locked again
2. The vesting period can be extended arbitrarily, delaying when users receive their full allocation
3. Trust in the protocol is undermined if vesting parameters are modified post-deployment without user consent

Recommendation

Add a check to ensure the current `startTime` has not yet been reached:

```
function setTime(uint40 _startTime, uint40 _endTime) public onlyOwner {
    if (block.timestamp > startTime) revert VestingAlreadyStarted();
    if (block.timestamp > _startTime) revert VestingAlreadyStarted();
    if (_startTime >= _endTime) revert VestingEndTimeError();

    startTime = _startTime;
    endTime = _endTime;

    emit VestingTimeSet(_startTime, _endTime);
}
```

Team Response

Acknowledged.

[L-02] Users Can Bypass Vesting During Initialization

Severity

Low Risk

Description

The contract is deployed in an unpaused state by default, with `pausedFlag` initialized to `0` and vesting times (`startTime` and `endTime`) initialized to `0`. This creates a vulnerability window during deployment where users who possess valid Merkle proofs can register and immediately claim their full allocation before vesting parameters are properly configured.

The typical initialization sequence, as can be seen in `test/S2Airdrop.t.sol`: 1. Deploy the contract (unpaused by default) 2. Call `setMerkleRoot()` to configure eligible addresses 3. Call `setTime()` to establish vesting parameters

After step 2 completes but before step 3 is executed, users can exploit this window to bypass vesting entirely. The `_vestedAmount()` function returns the full allocation when `startTime` is `0`:

```
if (block.timestamp < startTime) return 0;
```

Since `block.timestamp` is never less than `0`, this condition fails, and with `endTime` also being `0`, the function proceeds to calculate vested amounts incorrectly or returns unintended values.

Location of Affected Code

File: src/S2Airdrop.sol#L122-L128

```
constructor(address _owner, address _treasuryWallet) {
    if (_owner == address(0) || _treasuryWallet == address(0)) {
        revert ZeroAddress();
    }
    _initializeOwner(_owner);
    treasuryWallet = _treasuryWallet;
}
```

File: src/S2Airdrop.sol#L285-L301

```
function _vestedAmount(address user) internal view returns (uint256) {
    Airdrop memory details = airdropDetails[user];

    if (block.timestamp < startTime) return 0;

    uint256 leftVestedAmount = details.vestedAmount - details.
       spendLockedAmount;

    if (block.timestamp >= endTime) return leftVestedAmount;

    uint256 elapsed = block.timestamp - startTime;
    uint256 duration = endTime - startTime;

    return (leftVestedAmount * elapsed) / duration;
}
```

Impact

Users who obtain Merkle proofs (legitimately or through information leakage) can register and claim their entire token allocation immediately upon Merkle root deployment, completely bypassing the intended vesting mechanism. However, the likelihood of users obtaining valid Merkle proofs before initialization is complete is very low, as proofs are typically distributed after the contract is fully configured. This low likelihood is the primary reason this issue is rated as Low Risk.

Despite the low probability, exploitation could result in:

1. Premature distribution of tokens meant to vest over time
2. Loss of control over the token release schedule
3. Potential market impact from unexpected token circulation
4. Unfair advantage for users monitoring the mempool who can front-run initialization transactions

Recommendation

1. Deploy the contract in a paused state by initializing `pausedFlag = 1` in the constructor:

```
constructor(address _owner, address _treasuryWallet) {
    if (_owner == address(0) || _treasuryWallet == address(0)) {
        revert ZeroAddress();
    }
    _initializeOwner(_owner);
    treasuryWallet = _treasuryWallet;
    pausedFlag = 1;
}
```

2. Add validation checks for zero timestamps in both `claim()` and `_vestedAmount()` functions:

```
function _vestedAmount(address user) internal view returns (uint256) {
    if (startTime == 0 || endTime == 0) return 0;

    Airdrop memory details = airdropDetails[user];

    if (block.timestamp < startTime) return 0;

    uint256 leftVestedAmount = details.vestedAmount - details.
       spendLockedAmount;

    if (block.timestamp >= endTime) return leftVestedAmount;

    uint256 elapsed = block.timestamp - startTime;
    uint256 duration = endTime - startTime;

    return (leftVestedAmount * elapsed) / duration;
}
```

3. Follow a secure initialization sequence:

   - Deploy contract (automatically paused)
   - Set vesting times

- Set merkle root
- Fund the contract
- Unpause to enable user interactions

Team Response

Acknowledged.

[L-03] Merkle Root Is Mutable Mid-Airdrop

Severity

Low Risk

Description

The owner can change `merkleRoot` at any time. Already-registered users remain unaffected (state stored), but future registrations switch to the new root, which can change eligibility without warning.

Location of Affected Code

File: src/S2Airdrop.sol#L160-L163

```
function setMerkleRoot(bytes32 root) public onlyOwner {
    merkleRoot = root;
    emit MerkleRootSet(root);
}
```

Impact

Trust/governance risk: operator can exclude/replace users mid-campaign; users and integrators cannot assume a stable eligibility set after launch.

Recommendation

Freeze the root after a start time, or gate changes behind a timelock/multisig and communicate immutability guarantees. Optionally add a phase ID and store it to make root changes explicit on-chain.

Team Response

Acknowledged.

[I-01] Hardcoded Token Address Limits Deployment To A Specific Chain

Severity

Informational Risk

Description

The contract uses a hardcoded token address `0x133879524DDb38582cf0b93D10aDB789601Ff397` as a constant, which corresponds to the BORNE ERC-20 token on Avalanche mainnet. This implementation prevents the contract from being deployed on other blockchain networks where the BORNE token exists at different addresses.

For example, the BORNE token address on Abstract chain is s `0xcfd7cfeacc783a0d48ce076922b9d87aab2c4c1b`, making the current contract incompatible with that network without modifying the contract code.

Location of Affected Code

File: src/S2Airdrop.sol#L93

```
address private constant TOKEN = address(0
    x133879524DDb38582cf0b93D10aDB789601Ff397);
```

Impact

The hardcoded token address restricts deployment flexibility and requires separate contract implementations for each blockchain network. However, this configuration is acceptable if the protocol plans to deploy exclusively on the Avalanche mainnet.

Recommendation

Make the token address configurable by setting it in the constructor as an immutable variable.

Team Response

Acknowledged.

[I-02] No Recovery Path for non- `TOKEN` ERC-20s Sent to the Contract

Severity

Informational Risk

Description

The `withdrawTokens()` only transfers the hardcoded `TOKEN`. Any other ERC-20 sent by mistake is stuck forever; there's no generic sweep/recovery.

Location of Affected Code

File: src/S2Airdrop.sol#L276-L278

```
address private constant TOKEN = 0x1338...f397; // Only this token can be
    withdrawn:

function withdrawTokens(address to, uint256 amount) public onlyOwner {
    SafeTransferLib.safeTransfer(TOKEN, to, amount);
}
```

Impact

Permanent loss of mistakenly sent assets (common operational error during integrations / manual ops).

Recommendation

Add an owner-only `recoverERC20(address token, address to, uint256 amount)` guarded by appropriate policy (and events). Keep `TOKEN` logic unchanged for airdrop flows.

Team Response

Acknowledged.

[I-03] Missing Dedicated Event for Owner Withdrawals (Monitoring Blind Spot)

Severity

Informational Risk

Description

Owner withdrawals emit only the ERC-20 `Transfer` event. There's no contract-level event to reliably monitor admin outflows.

Location of Affected Code

File: src/S2Airdrop.sol#L276-L278

```
function withdrawTokens(address to, uint256 amount) public onlyOwner {
    SafeTransferLib.safeTransfer(TOKEN, to, amount); // no event
}
```

Recommendation

Emit `event TokensWithdrawn(address indexed to, uint256 amount);` inside `withdrawTokens()` and use it for monitoring and forensics.

Team Response

Acknowledged.

our shielding · Your smart contracts, our shielding · Your smart c

shieldify

Thank you!