shieldify

Honeypot Finance

NFT Staking

SECURITY REVIEW

Date: 7 October 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 has secured multiple millions in TVL across protocols, also can audit codebases written in Solidity, Rust, Go, Vyper, Move and Cairo.

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 Honeypot Finance - NFT Staking

This repository implements an NFT staking protocol where users can stake ERC-721 NFTs to earn ERC-20 reward tokens. Stakers accrue rewards over time, with a multiplier for longer staking durations. Users may also burn their staked NFTs for an additional bonus.

Features

- **NFT Staking:** Stake ERC-721 NFTs in NFTStaking to earn rewards.
- **Reward Token:** ERC-20 token (RewardsToken) minted by the staking contract.
- **Claiming Rewards:** Rewards accrue as:

$$rewardRatePerSecond \times \Delta t \times multiplier(elapsed), \text{with } multiplier = 1 + \lfloor elapsed/30\,days \rfloor.$$

- **Burn Bonus:** Burn staked NFTs for an extra bonus ($burnBonusBps$ basis points).
- **Security:** Uses OpenZeppelin's **ReentrancyGuard**, safe transfers, and owner-managed parameters.
- **Upgradeable Parameters:** Owner can update reward rate and burn bonus.

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 by 2 researchers from the Shieldify team.

Overall, the code is well-written. The audit report contributed by identifying one High, one Medium, and three Low severity issues. The findings are mainly related to incorrect burn reward multiplier reset by claiming and an unfair reward distribution.

The Honeypot team has done an excellent job on the development, demonstrating both expertise and dedication.

5.1 Protocol Summary

Project Name	**Honeypot Finance - NFT Staking**
Repository	new_nft_staking
Type of Project	ERC-20, ERC-721, NFT, Staking
Audit Timeline	2 days
Review Commit Hash	02d5afe0b5327642f48fffe29ec2cb4607a078e7
Fixes Review Commit Hash	6ca2b23dee18891692e9a95668719dce36e6fc35

5.2 Scope

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

File	nSLOC
src/NFTStaking.sol	145
src/RewardsToken.sol	22
Total	**167**

6. Findings Summary

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

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

ID	Title	Severity	Status
[H-01]	Incorrect Burn Reward Multiplier Reset on `claim()` Reduces Accrual	High	Fixed
[M-01]	Retroactive Parameter Updates Cause Unfair Reward Distribution	Medium	Fixed
[L-01]	Unrestricted `ERC721` Receipt Can Trap NFTs in the Contract	Low	Acknowledged
[L-02]	No Upper Bound on `rewardRatePerSecond()` Function	Low	Acknowledged
[L-03]	Re-minting Same `tokenId` After Burn Is Currently Possible	Low	Acknowledged
[I-01]	Integer Division Dust – Small `WEI` Truncation in Reward Math	Info	Acknowledged
[I-02]	Deduplicate Reward Calculation Logic to Prevent Drift	Info	Fixed
[I-03]	Inconsistent Mint Recipient in Burn Claims Risks Future Misdirection	Info	Fixed

7. Findings

[H-01] Incorrect Burn Reward Multiplier Reset on `claim()` Reduces Accrual

Severity

High Risk

Description

Burn-mode rewards compute the multiplier from time since the last burn claim and then reset that reference timestamp on each claim, erasing duration and keeping the multiplier near 1. Normal staking keeps the multiplier based on total elapsed since staking, so burn-mode accrual is materially underpaid relative to the intended "longer duration higher rewards" behavior.

Location of Affected Code

File: src/NFTStaking.sol#L161

```
function _claim(uint256 tokenId) internal returns (uint256 amount) {
  // code
  if (s.burned) {
      // Handle burn bonus claim
      require(s.burnedAt != 0, "INVALID_BURN_STATE");

      uint256 sinceBurn = block.timestamp - uint256(s.burnedAt);
      if (sinceBurn == 0) return 0;

      uint256 mBurn = _multiplier(sinceBurn);
      amount =
          (rewardRatePerSecond * sinceBurn * mBurn * burnBonusBps) /
          (ONE * MAX_BPS);

      if (amount == 0) return 0;

      // Update last burn claim time (resets multiplier reference)
      s.burnedAt = uint64(block.timestamp);

      // Mint burn bonus rewards
      rewards.mint(msg.sender, amount);
      emit BurnRewardClaimed(msg.sender, tokenId, amount);

      return amount;
  }

  // Normal claim path: multiplier uses total elapsed since stakedAt and
     does not reset
  uint256 nowTs = block.timestamp;
  if (nowTs <= s.lastClaimAt) return 0;

  uint256 delta = nowTs - uint256(s.lastClaimAt);
  uint256 elapsed = nowTs - uint256(s.stakedAt);
  uint256 m = _multiplier(elapsed);

  amount = (rewardRatePerSecond * delta * m) / ONE;

  if (amount == 0) return 0;

  // Update last claim time
  s.lastClaimAt = uint64(nowTs);

  // Mint rewards
  rewards.mint(s.owner, amount);
  emit RewardClaimed(s.owner, tokenId, amount);
  return amount;
}
```

Impact

Users receive significantly fewer rewards in burn mode than intended. Each claim truncates the multiplier accumulation window, preventing duration-based scaling comparable to normal staking. Burn-mode users are prevented from claiming rewards to avoid a multiplier reset, which puts them at a disadvantage compared to normal staking users.

Recommendation

Track two timestamps for burn mode: a fixed `burnStartAt` used only for multiplier calculation and a separate `lastBurnClaimAt` used for the delta window. Compute `mBurn` from `now - burnStartAt`, compute `delta` from `now - lastBurnClaimAt`. Alternatively, consider reusing the existing `lastClaimAt` timestamp instead of introducing a new field `lastBurnClaimAt`.

Team Response

Fixed.

[M-01] Retroactive Parameter Updates Cause Unfair Reward Distribution

Severity

Medium Risk

Description

Changing `rewardRatePerSecond` or `burnBonusBps` immediately affects accrual for the entire unclaimed interval, applying new parameters retroactively to past time. Rewards are computed at claim time using current values without epoching or checkpointing. This creates unfair treatment where users who claim before parameter changes receive different rewards than those who claim after, even for the same time periods.

Location of Affected Code

File: src/NFTStaking.sol#L161

```
function _claim(uint256 tokenId) internal returns (uint256 amount) {
  // code
  // Normal staking claim evaluates the entire delta with current
     parameters
  uint256 nowTs = block.timestamp;
  if (nowTs <= s.lastClaimAt) return 0;

  uint256 delta = nowTs - uint256(s.lastClaimAt);
  uint256 elapsed = nowTs - uint256(s.stakedAt);
  uint256 m = _multiplier(elapsed);

  amount = (rewardRatePerSecond * delta * m) / ONE;
  // code
}
```

File: src/NFTStaking.sol#L124

```
// Owner can change parameters at any time
function setParameters(
    uint256 _ratePerSecond,
    uint256 _burnBonusBps
) external onlyOwner {
    require(_burnBonusBps <= MAX_BPS, "BPS_EXCEEDS_MAX");
    require(_ratePerSecond > 0, "ZERO_RATE");
    rewardRatePerSecond = _ratePerSecond;
    burnBonusBps = _burnBonusBps;
    emit ParametersUpdated(_ratePerSecond, _burnBonusBps);
}
```

Impact

Elapsed time prior to the parameter change is mispriced at the new rate/bonus. Depending on direction, users may be overpaid or underpaid, and updates can be timed to skew payouts.

Proof of Concept

Example 1: Rewards increased 2x

- Initial `rewardRatePerSecond = 1e18` (1 token per second)
- Both UserA and UserB stake at timestamp 0
- Day 10: UserA claims receives 10 days × 1e18 = 864,000e18 tokens
- Day 10: Owner changes `rewardRatePerSecond = 2e18` (2 tokens per second)

- Day 20: UserA claims again receives 10 days × 2e18 = 1,728,000e18 tokens
- Day 20: UserB claims once receives 20 days × 2e18 = 3,456,000e18 tokens

Result: User A gets 2,592,000e18 total, UserB gets 3,456,000e18 total. User B receives 33% more rewards for the same 20-day period.

Example 2: Rewards decreased 2x

- Initial `rewardRatePerSecond = 2e18` (2 tokens per second)
- Both UserA and UserB stake at timestamp 0
- Day 10: UserA claims receives 10 days × 2e18 = 1,728,000e18 tokens
- Day 10: Owner changes `rewardRatePerSecond = 1e18` (1 token per second)
- Day 20: UserA claims again receives 10 days × 1e18 = 864,000e18 tokens
- Day 20: UserB claims once receives 20 days × 1e18 = 1,728,000e18 tokens

Result: UserA gets 2,592,000e18 total, UserB gets 1,728,000e18 total. UserA receives 50% more rewards for the same 20-day period.

Recommendation

Introduce parameter epochs with timestamps and compute piecewise across epochs intersecting `[lastClaimAt, now]`, or force a global checkpoint before parameter changes.

As a simpler alternative, store per-stake last-parameter values and pro-rate up to a recorded `parametersUpdatedAt` boundary.

Team Response

Fixed.

[L-01] Unrestricted ERC721 Receipt Can Trap NFTs in the Contract

Severity

Low Risk

Description

The contract accepts any ERC-721 via `onERC721Received()`, but only manages and returns the configured `nft` and only those staked via the `stake()` function. NFTs sent directly are accepted and then become unrecoverable because there is no withdrawal path for arbitrary token collections.

Location of Affected Code

File: src/NFTStaking.sol#L252

```
// Required for receiving ERC721 tokens
function onERC721Received(
    address,
    address,
    uint256,
    bytes calldata
) external pure override returns (bytes4) {
    return IERC721Receiver.onERC721Received.selector;
}
```

Impact

NFT tokens sent directly can become permanently locked in the contract, resulting in user loss and reputational risk.

Recommendation

Restrict `onERC721Received()` to only accept tokens from the configured `NFT` address during staking and revert otherwise, or implement an explicit rescue/withdraw path for non-whitelisted collections.

Alternatively, remove `onERC721Received()` and use `transferFrom()` instead of `safeTransferFrom()` during the staking call.

Team Response

Acknowledged.

[L-02] No Upper Bound on `rewardRatePerSecond()` Function

Severity

Low Risk

Description

The `setParameters()` and `initialize()` functions only enforce `_ratePerSecond > 0` and do not limit how large `rewardRatePerSecond` can be. Since reward calculation multiplies `rewardRatePerSecond * delta * m` before dividing, there may be an edge case where the owner can set an arbitrarily large `_ratePerSecond` and immediately cause intermediate multiplication overflow in `_claim()`. The `_claim()` function will then always revert, which will block claiming, unstaking and burning, leaving NFTs and rewards unusable (DoS). In general, adding an upper bound limit for `rewardRatePerSecond` would be a good practice.

Location of Affected Code

File: src/NFTStaking.sol#L66

```
function initialize(
    IERC721 _nft,
    RewardsToken _rewards,
    uint256 _ratePerSecond,
    uint256 _burnBonusBps,
    address initialOwner
) public initializer {
    require(address(_nft) != address(0), "ZERO_NFT");
    require(address(_rewards) != address(0), "ZERO_REWARDS");
    require(_burnBonusBps <= MAX_BPS, "BPS_EXCEEDS_MAX");
    require(_ratePerSecond > 0, "ZERO_RATE");

    __ReentrancyGuard_init();
    __Ownable_init(initialOwner);
    __UUPSUpgradeable_init();

    nft = _nft;
    rewards = _rewards;
    rewardRatePerSecond = _ratePerSecond;
    burnBonusBps = _burnBonusBps;

    emit ParametersUpdated(_ratePerSecond, _burnBonusBps);
}
```

File: src/NFTStaking.sol#L124

```
// Update parameters (only owner)
function setParameters(
    uint256 _ratePerSecond,
    uint256 _burnBonusBps
) external onlyOwner {
    require(_burnBonusBps <= MAX_BPS, "BPS_EXCEEDS_MAX");
    require(_ratePerSecond > 0, "ZERO_RATE");

    rewardRatePerSecond = _ratePerSecond;
    burnBonusBps = _burnBonusBps;

    emit ParametersUpdated(_ratePerSecond, _burnBonusBps);
}
```

Recommendation

Add upper bound limit for `rewardRatePerSecond()` state variable in `initialize()` and `setParameters()` functions.

Team Response

Acknowledged.

[L-03] Re-minting Same `tokenId` After Burn Is Currently Possible

Severity

Low Risk

Description

The `burn()` marks `stakes[tokenId].burned = true` and sets `burnedAt`. ERC-721 `_burn()` sets owner to `address(0)`, and standard `_mint()` allows minting tokenId again. That means the NFT collection may later re-mint the same `tokenId` to someone else.

The staking contract retains a `stake()` record with `burned = true` and `owner = original staker`. A new owner of a re-minted `tokenId` has no stake record. The original staker will continue to collect burn-phase rewards even though the token was re-minted to someone else. This creates a state inconsistency/design mismatch that you should document and decide on.

Location of Affected Code

File: src/NFTStaking.sol#L232

```
// Burn staked NFT and start earning burn bonus
function burn(uint256 tokenId) external nonReentrant {
    StakeData storage s = stakes[tokenId];
    require(s.owner == msg.sender, "NOT_OWNER");
    require(!s.burned, "ALREADY_BURNED");
    require(s.stakedAt != 0, "NOT_STAKED");

    // IMPORTANT: Claim all pending normal rewards before burning
    _claim(tokenId);

    // Burn the NFT
    IERC721Burnable(address(nft)).burn(tokenId);

    // Mark as burned and set burn timestamp
    s.burned = true;
    s.burnedAt = uint64(block.timestamp);

    emit Burned(msg.sender, tokenId);
}
```

Recommendation

Document that staking assumes tokenIds are unique forever and must never be re-minted or add detection/guard: - block `mint()` of any `tokenId` that previously had a stake record (requires co-ordination with NFT contract) or - on burn, delete the stake and emit a one-time burn reward instead of leaving a perpetual burn-state, or cap burn reward time.

Team Response

Acknowledged.

[I-01] Integer Division Dust - Small WEI Truncation in Reward Math

Severity

Informational Risk

Description

Reward math uses integer division and floors results. Small per-claim "dust" (wei) is lost. Example: `1e18 / 86400` truncates, costing ~6,400 wei/day for a 1 token/day rate (1 token/day rate is set in the tests). This affects accounting precision.

Location of Affected Code

File: src/NFTStaking.sol#L174

```
// rewardRatePerSecond set in initialize:
rewardRatePerSecond = _ratePerSecond; // e.g., 1e18 / 86400

// claim math:
uint256 amount = (rewardRatePerSecond * delta * m) / ONE;

// burn math:
amount = (rewardRatePerSecond * sinceBurn * mBurn * burnBonusBps) / (ONE
    * MAX_BPS);
```

Recommendation

Minimize dust by doing multiplication first with safe `mulDiv` and performing division as late as possible:

```
// compute t = rate * delta safely
uint256 t = FullMath.mulDiv(rewardRatePerSecond, delta, 1);
// amount = (t * m) / ONE safely
uint256 amount = FullMath.mulDiv(t, m, ONE);
```

Team Response

Acknowledged.

[I-02] Deduplicate Reward Calculation Logic to Prevent Drift

Severity

Informational Risk

Description

Reward and multiplier computations are duplicated across preview and claim paths for both normal and burn modes. Duplicate logic raises the risk of future divergence between previews and actual payouts during upgrades.

Location of Affected Code

File: src/NFTStaking.sol#L96

```solidity
// Preview pending rewards for a token
function previewPayout(uint256 tokenId) external view returns (uint256) {
    StakeData memory s = stakes[tokenId];
    if (s.owner == address(0)) return 0;

    if (s.burned) {
        // Calculate burn bonus accrual since last burn claim
        if (s.burnedAt == 0) return 0;
        uint256 sinceBurn = block.timestamp - uint256(s.burnedAt);
        if (sinceBurn == 0) return 0;
        uint256 mBurn = _multiplier(sinceBurn);
        // reward = rate * sinceBurn * mBurn scaled by burnBonusBps
        return
            (rewardRatePerSecond * sinceBurn * mBurn * burnBonusBps) /
            (ONE * MAX_BPS);
    }

    // Calculate normal staking rewards
    uint256 nowTs = block.timestamp;
    if (nowTs <= s.lastClaimAt) return 0;

    uint256 delta = nowTs - uint256(s.lastClaimAt);
    uint256 elapsed = nowTs - uint256(s.stakedAt);
    uint256 m = _multiplier(elapsed);

    return (rewardRatePerSecond * delta * m) / ONE;
}
```

File: src/NFTStaking.sol#L161

```
// Handle normal staking claim
function _claim(uint256 tokenId) internal returns (uint256 amount) {
    StakeData storage s = stakes[tokenId];
    require(s.owner == msg.sender, "NOT_OWNER");
    require(s.stakedAt != 0, "NOT_STAKED");

    if (s.burned) {
        // Handle burn bonus claim
        require(s.burnedAt != 0, "INVALID_BURN_STATE");

        uint256 sinceBurn = block.timestamp - uint256(s.burnedAt);
        if (sinceBurn == 0) return 0;

        uint256 mBurn = _multiplier(sinceBurn);
        amount =
            (rewardRatePerSecond * sinceBurn * mBurn * burnBonusBps) /
            (ONE * MAX_BPS);

        if (amount == 0) return 0;

        // Update last burn claim time
        s.burnedAt = uint64(block.timestamp);

        // Mint burn bonus rewards
        rewards.mint(msg.sender, amount);
        emit BurnRewardClaimed(msg.sender, tokenId, amount);

        return amount;
    }

    // Handle normal staking claim
    uint256 nowTs = block.timestamp;
    if (nowTs <= s.lastClaimAt) return 0;

    uint256 delta = nowTs - uint256(s.lastClaimAt);
    uint256 elapsed = nowTs - uint256(s.stakedAt);
    uint256 m = _multiplier(elapsed);

    amount = (rewardRatePerSecond * delta * m) / ONE;

    if (amount == 0) return 0;

    // Update last claim time
    s.lastClaimAt = uint64(nowTs);

    // Mint rewards
    rewards.mint(s.owner, amount);
    emit RewardClaimed(s.owner, tokenId, amount);

    return amount;
}
```

Impact

Inconsistent or incorrect rewards between preview and actual minting, undermining user expectations and potentially causing over-/under-payments.

Recommendation

Reuse `previewPayout()` in the claim path.

Team Response

Fixed.

[I-03] Inconsistent Mint Recipient in Burn Claims Risks Future Misdirection

Severity

Informational Risk

Description

Normal claims mint to the recorded stake owner, while burn claims mint to `msg.sender`. Although currently gated by `require(s.owner == msg.sender)`, future changes (e.g., permissionless claims) could direct rewards to an arbitrary caller instead of the rightful owner. Because the protocol is UUPS-upgradeable, any future implementation that relaxes claim access (permissionless/delegated/batch/relayed claims) can turn this inconsistency into a concrete payout-redirection bug.

Location of Affected Code

File: src/NFTStaking.sol#L206

```
function _claim(uint256 tokenId) internal returns (uint256 amount) {
  // code
  if (s.burned) {
    // code
    // Mint burn bonus rewards
    rewards.mint(msg.sender, amount);
    emit BurnRewardClaimed(msg.sender, tokenId, amount);
  }
  // Mint rewards
  rewards.mint(s.owner, amount);
  emit RewardClaimed(s.owner, tokenId, amount);
}
```

Impact

Rewards could be stolen or misdirected if a future UUPS upgrade enables permissionless, delegated, automated, or relayed claiming without aligning payout recipient semantics. Specifically, the burn-claim path paying `msg.sender` would let any caller siphon rewards that should go to the recorded stake owner.

Recommendation

Standardize on minting to `s.owner` in all claim paths.

Team Response

Fixed.

shieldify

Thank you!