hexens x Shib.io

Security Review Report for Shib.io

November 2025

Table of Contents

1. About Hexens
2. Executive summary
3. Security Review Details
    - Security Review Lead
    - Scope
    - Changelog
4. Severity Structure
    - Severity characteristics
    - Issue symbolic codes
5. Findings Summary
6. Weaknesses
    - New SOU claims/split/merged NFTs can execute previous payouts
    - Incorrect Decimal Scaling in usdValue calculation Causes Wrong USD Valuation
    - Overpayment/Underpayment When Payment Token Differs from Target Token
    - createPayout underfunds token transfers
    - Incorrect Claim Calculation Due to Dynamic Total Tokens Lost
    - Missing Token Funding in SOUDynamicPremiumStrategy Payout Creation
    - SOU Splitting Can Manipulate Equal-Share Rewards
    - SOUUniversalRewardStrategy _getTokenUSDValue Uses Invalid Snapshot ID
    - Bypassing Distribution Manager Allows Claims on NFTs You Don't Own
    - Donations Still Reduce Principal
    - DoS of executeClaim via split or merge
    - Previous distribution manager retains role
    - ownerOf can revert in batchExecuteClaims
    - Potential Original Owner Mismatch in handleBridgeCompensation
    - ERC721Enumerable Burn Reordering Breaks Deterministic SOU Token Handling
    - On-chain storage of user activities is expensive and inefficient

1. About Hexens

Hexens is a pioneering cybersecurity firm dedicated to establishing robust security standards for Web3 infrastructure, driving secure mass adoption through innovative protection technology and frameworks. As an industry elite experts in blockchain security, we deliver comprehensive audit solutions across specialized domains, including infrastructure security, Zero Knowledge Proof, novel cryptography, DeFi protocols, and NFTs.

Our methodology combines industry-standard security practices combined with unique methodology of two teams per audit, continuously advancing the field of Web3 security. This innovative approach has earned us recognition from industry leaders.

Since our founding in 2021, we have built an exceptional portfolio of enterprise clients, including major blockchain ecosystems and Web3 platforms.

2. Executive Summary

This report covers the security review for Shib. This review included the new SOU (Shib Owes You) contracts that implement a secondary NFT market around recovering and repayment from the Shibarium hack.

Our security assessment was a full review of the code, spanning a total of 1 week.

During our review, we identified 3 Critical severity vulnerabilities and 3 High severity vulnerabilities, which could have resulted in direct theft of user assets.

We also identified several minor severity vulnerabilities and code optimisations.

All of our reported issues were fixed or acknowledged by the development team and consequently validated by us.

We can confidently say that the overall security and code quality have increased after completion of our audit, however we do highly recommend to perform another security review.

3. Security Review Details

▪ Review Led by

Kasper Zwijsen, Head of Audits

▪ Scope

The analyzed resources are located on:

🔗 https://github.com/shibaone/SOU-contracts/tree/8ec25ae18336514cecedea6456b20df1a7c20b8e

The issues described in this report were fixed in the following commit:

🔗 https://github.com/shibaone/SOU-contracts/tree/559bdce35034819deecc78383d2f592d6bbf3c86

▪ Changelog

24 November 2025	Audit start
2 December 2025	Initial report
7 January 2026	Revision received
19 January 2026	Final report

4. Severity Structure

The vulnerability severity is calculated based on two components:

1. Impact of the vulnerability
2. Probability of the vulnerability

Impact	Probability			
	Rare	Unlikely	Likely	Very likely
Low	Low	Low	Medium	Medium
Medium	Low	Medium	Medium	High
High	Medium	Medium	High	Critical
Critical	Medium	High	Critical	Critical

- **Severity Characteristics**

Smart contract vulnerabilities can range in severity and impact, and it's important to understand their level of severity in order to prioritize their resolution. Here are the different types of severity levels of smart contract vulnerabilities:

Critical

Vulnerabilities that are highly likely to be exploited and can lead to catastrophic outcomes, such as total loss of protocol funds, unauthorized governance control, or permanent disruption of contract functionality.

High

Vulnerabilities that are likely to be exploited and can cause significant financial losses or severe operational disruptions, such as partial fund theft or temporary asset freezing.

Medium	Vulnerabilities that may be exploited under specific conditions and result in moderate harm, such as operational disruptions or limited financial impact without direct profit to the attacker.
Low	Vulnerabilities with low exploitation likelihood or minimal impact, affecting usability or efficiency but posing no significant security risk.
Informational	Issues that do not pose an immediate security risk but are relevant to best practices, code quality, or potential optimizations.

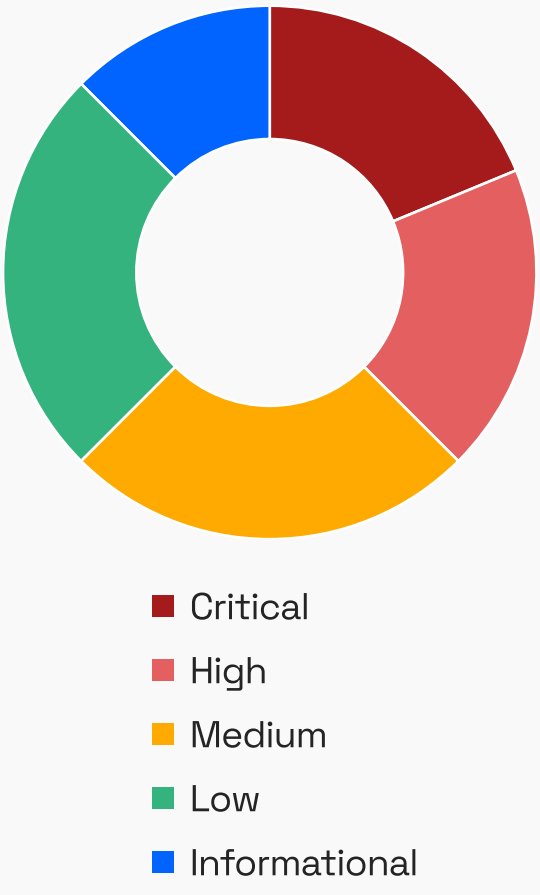
- **Issue Symbolic Codes**

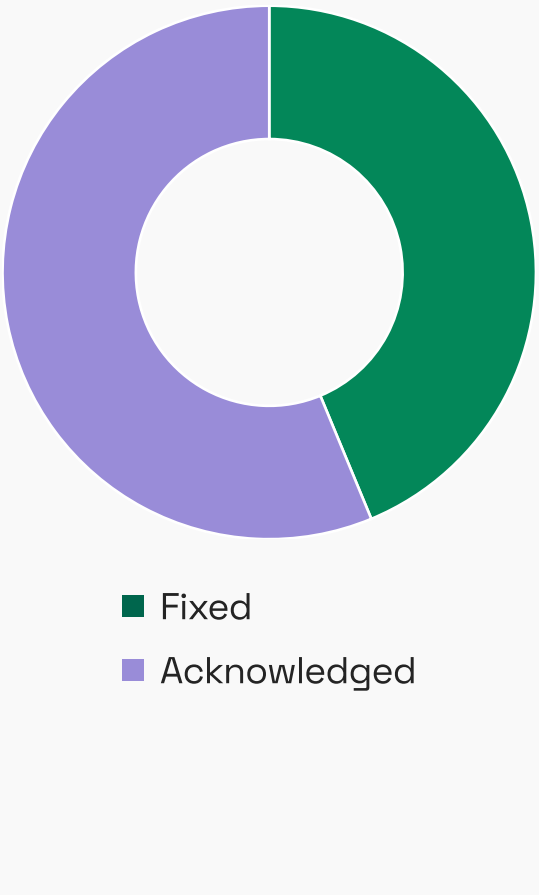
Each identified and validated issue is assigned a unique symbolic code during the security research stage.

Due to the structure of the vulnerability reporting flow, some rejected issues may be missing.

5. Findings Summary

Severity	Number of findings
Critical	3
High	3
Medium	4
Low	4
Informational	2
Total:	**16**





6. Weaknesses

This section contains the list of discovered weaknesses.

SHIB9-2 | New SOU claims/split/merged NFTs can execute previous payouts

Fixed

Severity:	Probability:	Impact:
Critical	Very likely	High

Path:

/contracts/SOUCore.sol
/contracts/strategies/*

Description:

The current **executeClaim** logic in **SOUUSDPayoutStrategy** allows a newly created claim in **SOUCore** to claim funds from a payout that was created before the claim existed. This could cause the rightful claimers from the previous payout unable to claim due to lack of funds.

Also, in the **SOUUniversalRewardStrategy**, it gives a fixed amount for each active SOU. If a NFT is split the same number of times as the active SOU tokens at the time of creation, the user can claim the entire payout.

Remediation:

Introduce a check in **SOUCore** so that when a strategy's **executeClaim** function calls **SOU**, it verifies whether the SOU was active before the time the payout was created. This ensures that only SOUs existing at payout creation can claim rewards.

The following actions update a SOU's **createdAt**. This timestamp should be compared with the payout's creation time to determine eligibility:

- Minting a SOU
- Merging SOUs
- Splitting a SOU

SHIB9-8 | Incorrect Decimal Scaling in usdValue calculation Causes Wrong USD Valuation

Fixed

Severity:	Probability:	Impact:
Critical	Very likely	High

Description:

The functions **_mintSOU()** and **handleBridgeCompensation()** for example compute USD value using:

```
(uint256 priceUSD,,) = priceOracle.getTokenPrice(token, snapshotId);
uint256 usdValue = (amount * priceUSD) / 1e18;
```

This assumes all tokens use 18 decimals, which is incorrect.

Why this is wrong

- Tokens used vary in decimals:
  - USDC = 6
  - USDT = 6
  - WBTC = 8
  - Dai = 18
- The oracle already stores token decimals, but the function ignores it.

As a result:

- Low-decimal tokens USD value becomes too small or even 0.
- high decimal tokens USD value becomes too large, allowing users to claim more USD than they should.

Example:

- Token: USDC (6 decimals)
- Amount to be compensated: 1,000,000 USDC
- Oracle price: 1 USDC = 1e8 USD units
- Expected USD: 1,000,000 USD value
- Actual USD: 100 (calculated as (**1e12 * 1e8) / 1e18**) USD value

This also happens in the strategy contracts, should be fixed everywhere.

Remediation:

Replace the entire USD calculation with the existing oracle function:

```
uint256 usdValue = priceOracle.calculateUSDValue(token, snapshotId, amount);
```

SHIB9-16 | Overpayment/Underpayment When Payment Token Differs from Target Token

Acknowledged

Severity:	Probability:	Impact:
Critical	Very likely	High

Path:

/contracts/strategies/SOUTokenSpecificPayoutStrategy.sol

Description:

The executeClaim function in **SOUTokenSpecificPayoutStrategy** calculates how much to pay a user (**tokenShare**) based on the amount of Target Token (Token A) they lost. But then it sends Payment Token (Token B) without adjusting for the price difference between Token A and Token B.

```
// tokenShare calculated based on Target Token lost
uint256 tokenShare = (remainingAmount * userTokensLost) / unclaimedTokenLost;

// USD reduction is correct
(uint256 referencePriceUSD,,) = priceOracle.getTokenPrice(payout.targetToken,
payout.referencePriceSnapshot);
uint256 usdValueReduction = (tokenShare * referencePriceUSD) / 1e18;

// Transfer Payment Token
IERC20(payout.paymentToken).transfer(recipient, tokenShare);
```

If Token A ≠ Token B in USD:

- User could receive too much or too little Token B.
- USD principal reduction is correct, but the actual token transfer does not match USD value.

Example:

- Token A = $2
- Token B = $1
- User lost 1 Token A
- The contract pays 1 Token B → the USD value of the transfer = $1
- But the principal reduction = $2 (should be 1$)

Remediation:

Adjust payouts to account for differences between target and payment tokens, ensuring the contract never sends more than it holds, and scale both the payout and USD principal reduction proportionally when funds are insufficient. Also, ensure that enough payment tokens are deposited during **createPayout** to cover expected claims.

SHIB9-3 | createPayout underfunds token transfers

Acknowledged

Severity:	Probability:	Impact:
High	Likely	High

Path:

/contracts/strategies/*

Description:

The **createPayout** function transfers **config.amount** of the ERC20 payment token to the contract, but **config.amount** is denominated in USD units, not token units. Later, during **executeClaim**, the payout is calculated in token units based on USD value and price snapshots. However the contract won't not hold enough tokens to cover the execute.

Remediation:

Convert the USD-denominated **config.amount** into actual token units (using the price snapshot) before transferring tokens from the creator, otherwise it will not have sufficient funds during execute.

```
++ uint256 tokenAmount = convertUSDToToken(config.amount, config.paymentToken,
config.priceSnapshotId);
++ IERC20(config.paymentToken).transferFrom(msg.sender, address(this), tokenAmount);
-- IERC20(config.paymentToken).transferFrom(msg.sender, address(this), config.amount);
```

SHIB9-10 | Incorrect Claim Calculation Due to Dynamic Total Tokens Lost

Fixed

Severity:	Probability:	Impact:
High	Likely	High

Path:

strategies/SOUTokenSpecificPayoutStrategy.sol

Description:

In the **SOUTokenSpecificPayoutStrategy**, the **executeClaim** function calculates claim amounts using the current total of tokens lost at the time of the claim:

```
uint256 totalTokenLost = _calculateTotalTokenLost(payout.targetToken);
uint256 unclaimedTokenLost = totalTokenLost - payout.claimedTokenLost;
uint256 tokenShare = (remainingAmount * userTokensLost) / unclaimedTokenLost;
```

Because **totalTokenLost** and **unclaimedTokenLost** change with each payout and any newly minted NFTs with token losses, the claim amounts vary depending on execution order. This can result in unexpected allocations.

Example:

- Payout: 1e18 tokens
- Three users each lost 1e18 tokens (total lost tokens 3e18)

Expected: **0.33e18** tokens per user
Actual: **[0.33e18, 0.4e18, 0.26e18]**

Remediation:

Potentially use a snapshot of total tokens lost at payout creation to calculate claim shares.

SHIB9-17 | Missing Token Funding in SOUDynamicPremiumStrategy Payout Creation

Acknowledged

Severity:	Probability:	Impact:
High	Likely	High

Path:

/contracts/SOUDynamicPremiumStrategy.sol

Description:

In **SOUDynamicPremiumStrategy**, the **createPayout** function stores the payout amount (**totalUSDValue**) and premium rules but does not transfer any ERC20 tokens to cover the payout unlike other strategies.

Claims later attempt to transfer tokens:

```
function executeClaim(uint256 tokenId, uint256 payoutId) external override  returns (uint256 claimedAmount) {
  << SNIP >>
  IERC20(payout.paymentToken).safeTransfer(recipient, tokenAmount);
```

Since the contract does not hold sufficient tokens, these transfers will revert, preventing users from claiming.

Remediation:

Ensure the contract always holds enough tokens to cover claims made by create payout, accounting for price changes and principal fluctuations.

SHIB9-5 | SOU Splitting Can Manipulate Equal-Share Rewards

Acknowledged

Severity:	Probability:	Impact:
Medium	Unlikely	Medium

Path:

strategies/SOUConditionalRewardStrategy.sol

Description:

In conditional reward strategy the claimable amount is calculated as **totalRewardAmount / eligibleCount**, where **eligibleCount** is the number of active SOUs with the any token loss. Users can call **splitSOU** to divide their SOU into multiple active SOUs. Each new SOU counts separately toward **eligibleCount**, allowing a single user to dilute rewards for others and capture a disproportionate portion of the reward.

Remediation:

The split function should correctly take these snapshotted amounts into account and not allow for creation of new eligible amounts. This could for example be done by forcing the claiming first.

SHIB9-6 | SOUUniversalRewardStrategy _getTokenUSDValue Uses Invalid Snapshot ID

Acknowledged

Severity:	Probability:	Impact:
Medium	Unlikely	Medium

Path:

strategies/SOUUniversalRewardStrategy.sol#L277-L285

Description:

The **_getTokenUSDValue** function in **SOUUniversalRewardStrategy** tries to get the latest price from **PriceOracle** by passing **bytes32(0)** as the snapshot ID. However, the **PriceOracle** does not handle **bytes32(0)** as a special case, so if no price exists for that snapshot, the function silently returns **0**.

```
try priceOracle.getTokenPrice(token, bytes32(0)) returns (uint256 priceUSD, uint256, uint256) {
    return (amount * priceUSD) / 1e18;
} catch {
    return 0;
}
```

Remediation:

Add a new function in **PriceOracle** to fetch the latest snapshot price for a token instead of passing bytes(0).

```
function _getTokenUSDValue(address token, uint256 amount) internal view returns (uint256) {
  bytes32 latestSnapshot = priceOracle.getLatestSnapshot(token);
  (uint256 priceUSD,,) = priceOracle.getTokenPrice(token, latestSnapshot);
  uint8 decimals = priceOracle.tokenDecimals(token);
  return (amount * priceUSD) / (10 ** decimals);
}
```

SHIB9-12 | Bypassing Distribution Manager Allows Claims on NFTs You Don't Own

Fixed

Severity:	Probability:	Impact:
Medium	Unlikely	Medium

Path:

contracts/SOUDistributionManager.sol

Description:

The **executeClaim** function in any of the strategy contracts is callable by any address. This allows a user to bypass the **SOUDistributionManager**, which normally enforces one check more than the execute claim's in the strategy:

```
// Verify ownership in DistributionManager
if (IERC721(address(souContract)).ownerOf(tokenId) != msg.sender) {
    revert DistributionManager__NotTokenOwner();
}
```

Because the check is performed only in the **DistributionManager**, a user can call **strategy.executeClaim()** directly on a token they do not own.

Additionally, if there's an activity tracker enabled or off-chain event tracked, it won't be recorded which could lead to accounting issues, because its only tracked in **SOUDistributionManager**:

```
function executeClaim(
    address strategy,
    uint256 payoutId,
    uint256 tokenId,
    uint8 activityType
) external override nonReentrant returns (uint256 amount) {
    << SNIP >>
    // Execute claim through strategy
    amount = strategyContract.executeClaim(tokenId, payoutId);

    // Record activity in ActivityTracker via SOUCore
    souContract.recordActivity(
        msg.sender,
        tokenId,
```

```
            activityType,
            abi.encode(strategy, payoutId, amount)
        );

        emit ClaimExecuted(strategy, payoutId, tokenId, msg.sender, amount);

        return amount;
    }
```

Remediation:

Restrict the **executeClaim** functions so they can only be called through the DistributionManager contract.

SHIB9-15 | Donations Still Reduce Principal

Acknowledged

Severity:	Probability:	Impact:
Medium	Likely	Medium

Description:

The **SOUDonationStrategy** currently reduces **currentPrincipalUSD** when a donation is executed:

```
souContract.reducePrincipal(tokenId, usdValue);
souContract.increasePaidOut(tokenId, usdValue);
```

This reduces the SOU token's principal when a donation is claimed.

- The interface and documentation state donations should not reduce principal.

```
### B. Donation Claims

**Mechanism:**
- Third parties donate to SOU holder pool
- Could be community, sponsors, or platform partners
- Distributed pro-rata to active token holders
- Does NOT reduce principal
- Increases `donationsReceivedUSD`
```

Remediation:

Instead of **reducePrincipal/increasePaidOut**, use the existing function:

```
souContract.increaseDonationsReceived(tokenId, usdValue);
```

SHIB9-9 | DoS of executeClaim via split or merge

Acknowledged

Severity:	Probability:	Impact:
Low	Rare	Medium

Description:

```
for (uint256 j = 0; j < lostTokens.length; j++) {
  address tokenAddr = lostTokens[j];
  uint256 amount = _tokenLosses[tokenIds[i]][tokenAddr];

  // Add to merged token (or create if first time)
  if (_tokenLosses[newTokenId][tokenAddr] == 0) {
      _lostTokens[newTokenId].push(tokenAddr);
      _tokenToHolders[tokenAddr].push(newTokenId);
  }
  _tokenLosses[newTokenId][tokenAddr] += amount;
}
```

Using the **mergeSOU** or **splitSOU** functions, the contract pushes **newTokenId** into the **_tokenToHolders** array.

However, even when a **tokenId** is removed, **_tokenToHolders** is never cleaned up, so the array continues growing indefinitely.

```
/**
* @notice Execute claim for a token
*/
function executeClaim(uint256 tokenId, uint256 payoutId) external override nonReentrant returns (uint256
claimedAmount) {
  TokenSpecificPayout storage payout = _payouts[payoutId];

  if (!payout.active) revert TokenSpecificStrategy__PayoutInactive();
  if (_claimed[payoutId][tokenId]) revert TokenSpecificStrategy__AlreadyClaimed();

  // Check eligibility
  if (!souContract.hasLostToken(tokenId, payout.targetToken)) {
      revert TokenSpecificStrategy__NotEligible();
  }

  uint256 userTokensLost = souContract.getTokenLost(tokenId, payout.targetToken);
```

```
    if (userTokensLost == 0) {
        revert TokenSpecificStrategy__InsufficientTokenRemaining();
    }

    // Get CURRENT total of target token lost (dynamic - changes as new SOUs mint)
    uint256 totalTokenLost = _calculateTotalTokenLost(payout.targetToken);
    ...
}
```

```
/**
* @notice Get all token IDs that lost specific token
*/
function getTokenIdsWithLoss(address token) external view override returns (uint256[] memory) {
    return _tokenToHolders[token];
}
```

If **_tokenToHolders** grows without limit, calling **getTokenIdsWithLoss** can consume unexpectedly large amounts of gas.

This may prevent the **executeClaim** function from being executed at all, effectively causing a denial-of-service (DoS) condition.

Remediation:

Unbounded arrays should be avoided and so the condition should be checked through other means.

SHIB9-1 | Previous distribution manager retains role

Fixed

Severity:	Probability:	Impact:
Low	Unlikely	Low

Path:

/contracts/SOUCore.sol

Description:

setDistributionManager() grants the new manager role but never revokes it from the previous one. As a result, all older distribution managers keep full authority to modify claims, reduce principal, and update payouts.

Remediation:

```
function setDistributionManager(address newManager) external onlyRole(ADMIN_ROLE) {
    if (newManager == address(0)) revert SOUCore__ZeroAddress();
    // Revoke role from previous manager if set
++  if (distributionManager != address(0)) {
++      revokeRole(DISTRIBUTION_MANAGER_ROLE, distributionManager);
++  }
    // Assign new manager
    distributionManager = newManager;
    grantRole(DISTRIBUTION_MANAGER_ROLE, newManager);
}
```

SHIB9-7 | ownerOf can revert in batchExecuteClaims

Fixed

Severity:	Probability:	Impact:
Low	Unlikely	Low

Path:

/contracts/SOUDistributionManager.sol

Description:

The function **batchExecuteClaims** attempts to skip claims for tokens the caller does not own:

```
// Skip if not owner
if (IERC721(address(souContract)).ownerOf(claim.tokenId) != msg.sender) {
    continue;
}
```

However, OpenZeppelin's ERC721 implementation (source) shows that **ownerOf** reverts if the token does not exist:

```
function _requireOwned(uint256 tokenId) internal view returns (address) {
    address owner = _ownerOf(tokenId);
    if (owner == address(0)) {
        revert ERC721NonexistentToken(tokenId);
    }
    return owner;
}
```

As a result, any nonexistent **tokenId** causes the transaction to revert, defeating the goal of skipping invalid claims.

Remediation:

Use the existing function:

```
if (!souContract.exists(claim.tokenId)) continue;
```

SHIB9-13 | Donations Still Reduce Principal

Acknowledged

Severity:	Probability:	Impact:
Low	Rare	Low

Description:

The function **handleBridgeCompensation** attempts to add new token losses to an existing SOU NFT if the user already owns at least one token:

```
uint256 userBalance = balanceOf(user);
if (userBalance == 0) {
    tokenId = _mintSOU(user, token, amount, snapshotId);
} else {
    tokenId = tokenOfOwnerByIndex(user, 0);
    // Add losses to this token
}
```

If the user already owns a token, the function simply adds the new losses to the first token they own. However, **_claims[tokenId].originalOwner** may reference a different user, because NFTs can be transferred.

If the first token was received via a tiny transfer (e.g., 1 wei) from another user, the **originalOwner** will differ. While not currently used, this could cause future issues if **originalOwner** is relied on.

Remediation:

If **originalOwner** were to be used, it should be used correctly when adding new losses to the NFT.

SHIB9-14 | ERC721Enumerable Burn Reordering Breaks Deterministic SOU Token Handling

Fixed

Severity:	Probability:	Impact:
Informational	Rare	Informational

Description:

When a user splits a SOU NFT, the contract mints two new tokens and burns the original. The **handleBridgeCompensation()** function adds new compensation to the "first SOU" owned by the user, determined by:

```
// User has SOU - get their first SOU and add to it
tokenId = tokenOfOwnerByIndex(user, 0);
```

Because **SOUCore** inherits from **ERC721EnumerableUpgradeable**, burning a token uses swap-and-pop logic. This reorders the **_ownedTokens** array for the user, making the last token the first token now:

```
function _removeTokenFromOwnerEnumeration(address from, uint256 tokenId) private {
    // To prevent a gap in from's tokens array, we store the last token in the index of the token to delete, and
    // then delete the last slot (swap and pop).

    uint256 lastTokenIndex = balanceOf(from);
    uint256 tokenIndex = _ownedTokensIndex[tokenId];

    mapping(uint256 index => uint256) storage _ownedTokensByOwner = _ownedTokens[from];

    // When the token to delete is the last token, the swap operation is unnecessary
    if (tokenIndex != lastTokenIndex) {
        uint256 lastTokenId = _ownedTokensByOwner[lastTokenIndex];

        _ownedTokensByOwner[tokenIndex] = lastTokenId; // Move the last token to the slot of the to-delete
token
        _ownedTokensIndex[lastTokenId] = tokenIndex; // Update the moved token's index
    }

    // This also deletes the contents at the last position of the array
    delete _ownedTokensIndex[tokenId];
    delete _ownedTokensByOwner[lastTokenIndex];
}
```

Scenario

1. User receives a minted token via **handleBridgeCompensation()** → token 1.
2. User calls **splitSOU(1, 50)** → contract mints token 2 and token 3 and burns token 1.
3. User gets another **handleBridgeCompensation()** → compensation is incorrectly added to token 3 instead of token 2.

Remediation:

Consider burning first before minting:

```
++ _burn(tokenId);
uint256 newTokenId1 = _createSplitToken(tokenId, firstAmount, owner);
uint256 newTokenId2 = _createSplitToken(tokenId, secondAmount, owner);
-- _burn(tokenId);
```

SHIB9-4 | On-chain storage of user activities is expensive and inefficient

Acknowledged

Severity:	Probability:	Impact:
Informational	Rare	Informational

Path:

/contracts/SOUActivityTracker.sol

Description:

The contract **SOUActivityTracker** can be used in the **SouCore** to store all user activities, however on-chain this is costly. Emitting events instead would be much cheaper, and off-chain programs can efficiently read and process the data. It would highly be recommended to not use on chain data storing.

```
function recordBridgedToken(
    address user,
    uint256 tokenId,
    address token,
    uint256 amount,
    bytes32 priceSnapshotId,
    uint256 usdValue
) external onlyRole(TRACKER_ROLE) whenNotPaused {
    require(user != address(0), "SOUActivityTracker: Invalid user");
    require(token != address(0), "SOUActivityTracker: Invalid token");

    BridgedToken memory bridgedToken = BridgedToken({
        token: token,
        amount: amount,
        priceSnapshotId: priceSnapshotId,
        usdValue: usdValue,
        timestamp: block.timestamp
    });

    userBridgedTokens[user].push(bridgedToken);

    emit BridgedTokenRecorded(user, tokenId, token, amount, usdValue);
}
```

Remediation:

Rely on events to handle full activity history off-chain.

hexens x Shib.io