# Audit Report – traitforge

Repository: [https://github.com/code-423n4/2024-07-traitforge]

Date: [29.12.2025]

Auditor: [Harisuthan]

## Summary:

Total issues found: [7 (high and medium included)]

Severity breakdown: [2 high, 5 medium]

## Issue H-1 : Unauthorized Caller Access Breaks Generation Increment

## Summary:

The `_incrementGeneration()` function in `TraitForgeNFT` is intended to increment the current generation and initialize entropy indices. However, it calls entropyGenerator.initializeAlphaIndices() without proper caller permissions. Since initializeAlphaIndices() is restricted to `onlyOwner`, this causes transactions to revert unexpectedly, breaking the minting process when max tokens per generation are reached.

## Vulnerability Detail:

`_incrementGeneration()` checks if the current generation has reached `maxTokensPerGen` and then increments `currentGeneration`.

After incrementing, it calls `entropyGenerator.initializeAlphaIndices()` to reset slot and number indices.

initializeAlphaIndices() in EntropyGenerator is restricted to onlyOwner, but _incrementGeneration() is called by the NFT contract during normal minting.

When minting triggers `_incrementGeneration()`, the call fails because the NFT contract is not the owner, reverting the transaction.

## Step-by-step example:

1. Mint tokens until generationMintCounts[currentGeneration] >= maxTokensPerGen.
2. _mintInternal() calls _incrementGeneration().
3. _incrementGeneration() calls entropyGenerator.initializeAlphaIndices().
4. Transaction reverts with Ownable: caller is not the owner.

## Code snippet proof:

▶

## Proof of code

▶ Click to view the test code

## Impact:

Normal minting may fail when the generation limit is reached, making the NFT contract unusable for further mints.

Users attempting to mint tokens during this stage will experience reverts, resulting in a denial of service.

## Recommendation:

Use a proper access control modifier like onlyAllowedCaller for _incrementGeneration().

Ensure _incrementGeneration() can safely call initializeAlphaIndices() without reverting for normal mint operations.

Add tests to confirm minting works when generation increments.

## Issue H-2 : Incorrect Maximum Token Check in `mintWithBudget()`

## Summary:

The `mintWithBudget()` function is intended to allow users to mint multiple tokens in a single transaction using their ETH budget. However, it checks _tokenIds < maxTokensPerGen instead of verifying generation-specific mint counts. This can block minting unexpectedly, preventing users from using the budget properly.

## Vulnerability Detail:

The function calculates mintPrice and loops while `budgetLeft >= mintPrice and _tokenIds < maxTokensPerGen.`

_tokenIds is a global counter for all tokens, not per-generation.

Once _tokenIds exceeds maxTokensPerGen, the while loop stops, even if the current generation has available mints.

This causes users to fail budgeted minting, forcing refunds for unused ETH.

## Step-by-step example:

1. Suppose maxTokensPerGen = 5 and _tokenIds = 6 but generationMintCounts[currentGeneration] = 2.
2. A user calls mintWithBudget() with enough ETH to mint multiple tokens.
3. The while loop fails because _tokenIds < maxTokensPerGen is false.

No new tokens are minted, and the remaining ETH is refunded.

## Code snippet proof:

▶

## Impact:

Users cannot mint with their budget once global _tokenIds exceeds maxTokensPerGen.

Budgeted minting becomes unusable, leading to poor user experience and unexpected refunds.

Could break contract functionality if _tokenIds grows faster than generation limits allow.

## Recommendation:

Replace _tokenIds < maxTokensPerGen with a generation-specific check, e.g., generationMintCounts[currentGeneration] < maxTokensPerGen.

Ensure budgeted minting loops correctly across the current generation.

Add tests for budgeted minting at generation boundaries to prevent reverts or failed mints.

# Issue M-1 : Mint Price Calculation Mismatch Across Generations

## Summary:

The `mintToken()` function calculates `mintPrice` using `calculateMintPrice()` before minting, but `_mintInternal()` may trigger `_incrementGeneration()` if the current generation reaches its max. As a result, users may pay the previous generation's price while receiving a token from the next generation, leading to overpayment.

## Vulnerability Detail:

`calculateMintPrice()` uses `generationMintCounts[currentGeneration]` to compute the price:

▶

`_mintInternal()` checks if the current generation is full:

▶

If the max is reached, `_incrementGeneration()` increments currentGeneration after the price is already calculated.

The minted token then belongs to the new generation, but the user has paid for the previous generation's price.

## Step-by-step example:

1. Current generation 1 has maxTokensPerGen - 1 tokens minted.
2. A user calls mintToken() and sends ETH for calculateMintPrice() of generation 1.
3. _mintInternal() mints the token, but since generation 1 reached max, _incrementGeneration() is triggered.

4. User receives a generation 2 token, paying generation 1 price.

## Test case proof:

▶

## Impact:

Users may overpay for NFTs, paying a lower generation price but receiving a higher generation token.

Can cause loss of user trust, complaints, and potential financial disputes.

## Recommendation:

Calculate mint price after checking if generation needs incrementing.

Ensure that price paid always matches the generation of the token received.

Add unit tests for edge cases when minting the last token of a generation

# Issue M-2 : Missing Pause/Unpause Functionality

## Summary:

The contract inherits Pausable but does not provide functions to pause or unpause the contract. As a result, the owner cannot stop critical operations in case of an emergency or exploit.

## Vulnerability Detail:

Pausable is imported and the whenNotPaused modifier is used in functions like mintToken() and mintWithBudget().

However, there is no public function to call pause() or unpause().

Without these functions, the owner cannot pause the contract if a bug or exploit is discovered.

## Impact:

In case of a critical bug, the owner cannot prevent users from interacting with the contract.

Could lead to loss of funds, tokens, or other unintended behavior during an emergency.

## Recommendation:

▶

## Impact:

Users may receive less Ether than expected due to frontrunning.

Can cause user dissatisfaction, complaints, or loss of trust.

Medium severity because it does not break the contract, but affects fund fairness.

## Recommendation:

Add a slippage tolerance parameter in `nuke()`, e.g., user specifies minExpectedAmount.

Check the calculated claimAmount against minExpectedAmount:

```
require(claimAmount >= minExpectedAmount, "Claim amount below acceptable
```

## Issue M-4 : Excess Funds Not Refunded in `forgeWithListed()`

## Summary:

The `forgeWithListed()` function requires users to pay a forging fee. However, if a user sends more ETH than required, the extra funds are not refunded, resulting in locked Ether within the contract.

## Vulnerability Detail:

The function checks that the sent value is at least the forging fee:

```
require(msg.value >= forgingFee, 'Insufficient fee for forging');
```

No code exists to refund the excess ETH.

Users can accidentally or intentionally send more than forgingFee, and the extra amount will remain locked in the contract indefinitely.

## Step-by-step example:

1. Forging fee = 0.5 ETH.
2. User sends 1 ETH for forging.

T3) ransaction succeeds, but 0.5 ETH excess remains in contract.

4. No mechanism exists to recover this excess, effectively locking user funds.

## Impact:

Users may lose Ether permanently if they overpay.

Could cause user dissatisfaction, complaints, and loss of trust.

Medium severity because it does not break functionality, but affects fund safety.

## Recommendation:

Refund excess ETH after fee is deducted

## Issue M-5 : Incorrect Age Calculation Rounds Down Small Durations

## Summary:

The `calculateAge()` function calculates token age in full days, rounding down partial days. This causes tokens less than a day old to have 0 age, which may incorrectly affect age-dependent calculations.

## Vulnerability Detail:

Current calculation:

```
uint256 daysOld = (block.timestamp - nftContract.getTokenCreationTimesta
```

This only counts complete 24-hour periods.

Partial days (hours or minutes) are ignored, rounding down to zero.

Functions using `calculateAge()` (e.g., nuke() or performance calculations) may understate age, giving lower rewards or affecting game mechanics.

## Step-by-step example:

1. Token created at timestamp t0.
2. Current time is 18 hours later: block.timestamp - t0 = 18 * 3600 = 64800 seconds.
3. Days calculation: 64800 / 60 / 60 / 24 = 0 (rounds down).
4. Age-dependent calculation (e.g., age = daysOld * perfomanceFactor * MAX_DENOMINATOR * ageMultiplier / 365) will also be zero, underestimating the token's true age.

## Code snippet proof:

uint256 daysOld = (block.timestamp - nftContract.getTokenCreationTimestamp(tokenId)) / 60 / 60 / 24; // Rounds down any duration less than 24 hours

## Impact:

1. Newly minted tokens (<1 day old) may have 0 age.
2. affect rewards, or age-based mechanics in the contract.

## Recommendation:

Use fractional age calculation instead of full days, e.g., calculate age in seconds or hours and convert with higher precision.