2024-07-traitforge (TraitForge)

Benchmark Security Review Performed by Bug Hunter

22.06.2025

Table of Contents

About Bug Hunter 3

Notices and Remarks 4

Executive Summary 5

Project Summary 6

Detailed Findings 7

1. Incorrect Mint Cap Logic Across Generations 7
2. Misconfiguration of 'taxCut' Can Lead to Division by Zero or Unfair Fee Distribution 9
3. Lack of Upper Bound on Generation Incrementation May Lead to Unbounded Growth or Overflow 11
4. Missing Access Control Allows Unrestricted Generation Incrementation 12
5. Funds Can Be Locked Due to Untrusted Ether Transfer in `nuke` Function 13
6. Lack of Slippage Protection in `mintWithBudget` May Lead to Fewer Mints Than Expected 15
7. Unsafe and Ineffective Pseudo-Randomness in `writeEntropyBatch1` 17
8. DAO Stage in `receive()` Function May Be Unreachable Due to Uncontrollable External State 19
9. Hardcoded Special Return Value Creates Hidden Control Flow and Reduces Entropy Quality 21
10. Division by Zero Risk and Unpredictable Age Due to Performance Factor from Entropy 23
11. Forger Can Forge More Times Than Intended by Relisting Between Calls 25
12. Lack of Slippage Protection in Token Nuking May Lead to Unexpected Claims 28
13. Excess ETH Sent to `forgeWithListed` May Become Permanently Stuck 30
14. Unused Variable `amountMinted` in `mintWithBudget` Function 33
15. Redundant Condition Can Be Simplified in `_mintNewEntity` 35
16. fetchListings Returns Array with Unused Zero Index and Possible Empty Entries 37
17. Seller Can Block NFT Sale by Reverting ETH Transfer 38
18. Ambiguous Modulus Usage in Entropy Generation 40

About Bug Hunter

Bug Hunter is an automated code-review tool for Solidity smart contracts, developed by Truscova GmbH. It uses proprietary algorithms and advanced code-analysis techniques to identify potential vulnerabilities efficiently.

Bug Hunter is designed to support developers in the Ethereum ecosystem by delivering consistent, scalable assessments that align with best practices in Web 3.0 security. Reports are generated via the Bug Hunter tool and are accessible to clients through a secure dashboard web interface or relevant Git repositories.

Bug Hunter is a product of Truscova, a security R&D company headquartered in Bremen, Germany. Founded in late 2022, Truscova specializes in formal verification, fuzzing, dynamic and static analysis, and security tooling. Its founding team includes researchers with 35+ books, several patents, and over 20 000 citations in software testing, verification, security, and machine learning.

To explore more about Bug Hunter, visit:

- Bug Hunter Website
- Bug Hunter Documentation
- Example Reports

To explore more about Truscova, visit:

- Truscova Website
- X
- LinkedIn

Notices and Remarks

Copyright and Distribution

This security-review report was generated with Truscova's Bug Hunter tool to benchmark its performance on selected NFT projects. It is made available through the Bug Hunter dashboard and relevant Git repositories. Users may choose to share or publish this report at their own discretion.

Test Coverage Disclaimer

This security review was conducted exclusively by Bug Hunter, an automated code-review tool for Solidity codebases. Bug Hunter uses a combination of proprietary algorithms and programmatic techniques to identify potential vulnerabilities in Solidity smart contracts.

The findings in this report should not be considered a complete or exhaustive list of all possible security issues in the reviewed codebase and are dependent on Bug Hunter's coverage of detector rules. The full list of rules covered by Bug Hunter automated code review is available at https://docs.bughunter.live/.

Executive Summary

Overview and Scope

A codebase from project 2024-07-traitforge was reviewed for security vulnerabilities by Bug Hunter with the following details:

- GitHub Repository: https://github.com/code-423n4/2024-07-traitforge
  Commit hash: `279b2887e3d38bc219a05d332cbcb0655b2dc644`

Bug Hunter reviewed the following files:

DevFund.sol EntityForging.sol EntityTrading.sol EntropyGenerator.sol NukeFund.sol TraitForgeNft.sol

Summary of Findings

The uncovered vulnerabilities identified during the security review are summarised in the table below:

Severity	Count
High	4
Medium	9
Low	5

Project Summary

Contact Information

For support, contact us at support@bughunter.live

Project Timeline

Date	Event
22.06.2025	Project submitted
22.06.2025	Security review concluded

Detailed Findings

1. Incorrect Mint Cap Logic Across Generations

Severity	High
Finding ID	BH-2024-07-traitforge-01
Target	contracts/TraitForgeNft/TraitForgeNft.sol
Function name	mintWithBudget

Description

The `mintWithBudget` function attempts to limit the number of tokens minted within a specific generation using the condition `while (budgetLeft >= mintPrice && _tokenIds < maxTokensPerGen)`. However, `_tokenIds` tracks the total number of tokens ever minted across all generations. This means that the minting limit is applied globally instead of per generation. As a result, if previous generations have minted tokens, users may be able to bypass generation-specific limits or be incorrectly prevented from minting even if the current generation has capacity.

```
function mintWithBudget(
    bytes32[] calldata proof
  )
    public
    payable
    whenNotPaused
    nonReentrant
    onlyWhitelisted(proof, keccak256(abi.encodePacked(msg.sender)))
  {
    uint256 mintPrice = calculateMintPrice();
    uint256 amountMinted = 0;
    uint256 budgetLeft = msg.value;

    while (budgetLeft >= mintPrice && _tokenIds < maxTokensPerGen) {
      _mintInternal(msg.sender, mintPrice);
      amountMinted++;
      budgetLeft -= mintPrice;
      mintPrice = calculateMintPrice();
    }
    if (budgetLeft > 0) {
      (bool refundSuccess, ) = msg.sender.call{ value: budgetLeft }('');
      require(refundSuccess, 'Refund failed.');
    }
  }
```

Recommendation

Introduce a separate counter or mapping to track the number of tokens minted per generation (e.g., `tokensMintedPerGen[currentGeneration]`) and update the condition to check that value against `maxTokensPerGen` . Ensure that the generation context is clearly defined and consistently used throughout the minting process.

2. Misconfiguration of 'taxCut' Can Lead to Division by Zero or Unfair Fee Distribution

Severity	High
Finding ID	BH-2024-07-traitforge-02
Target	contracts/EntityForging/EntityForging.sol
Function name	forgeWithListed

Description

The `forgeWithListed` function uses the variable `taxCut` to calculate the portion of the forging fee that goes to the developer fund and the forger owner. Specifically, it computes `devFee = forgingFee / taxCut`. If `taxCut` is ever set to zero, this will result in a division-by-zero error and revert the transaction. Furthermore, even when non-zero, incorrect `taxCut` values can result in severe misallocation of funds — either giving all the fee to the developer or none. This introduces risk of denial-of-service or economic exploitation, particularly if `taxCut` can be changed by an external admin or owner without constraints.

```
function forgeWithListed(
    uint256 forgerTokenId,
    uint256 mergerTokenId
  ) external payable whenNotPaused nonReentrant returns (uint256) {
    ...
    uint256 forgingFee = _forgerListingInfo.fee;
    require(msg.value >= forgingFee, 'Insufficient fee for forging');

    ...

    uint256 devFee = forgingFee / taxCut;
    uint256 forgerShare = forgingFee - devFee;
    address payable forgerOwner = payable(nftContract.ownerOf(forgerTokenId));

    ...
  }
```

Recommendation

Add a `require(taxCut > 0)` check before performing division to prevent division-by-zero reverts. Enforce reasonable bounds for `taxCut` (e.g., between 1 and 1000) to prevent extreme fee allocations. Consider emitting an event whenever `taxCut` is updated, and document its economic implications clearly. If possible, replace it with a fixed numerator/denominator fee system for greater clarity.

3. Lack of Upper Bound on Generation Incrementation May Lead to Unbounded Growth or Overflow

Severity	High
Finding ID	BH-2024-07-traitforge-03
Target	contracts/TraitForgeNft/TraitForgeNft.sol
Function name	_incrementGeneration

Description

The `_incrementGeneration` function increments the `currentGeneration` variable and resets the mint count for the new generation. However, it does not impose an upper limit on the number of generations that can be created. If `_incrementGeneration` is called repeatedly (intentionally or due to a bug elsewhere), it could result in unbounded generation growth, which may increase storage usage and gas costs. Additionally, if `currentGeneration` is a small unsigned integer (e.g., `uint8`), this could eventually lead to an overflow and wraparound, causing logical bugs in generation tracking and token pricing.

```
function _incrementGeneration() private {
    require(
      generationMintCounts[currentGeneration] >= maxTokensPerGen,
      'Generation limit not yet reached'
    );
    currentGeneration++;
    generationMintCounts[currentGeneration] = 0;
    priceIncrement = priceIncrement + priceIncrementByGen;
    entropyGenerator.initializeAlphaIndices();
    emit GenerationIncremented(currentGeneration);
  }
```

Recommendation

Introduce an upper limit (e.g., `maxGenerations`) and enforce it using `require(currentGeneration < maxGenerations)` before incrementing. Also, ensure `currentGeneration` is of a sufficiently large type (e.g., `uint256`) to avoid overflow. Consider validating whether price increments or entropy behavior should be capped or adjusted per generation.

4. Missing Access Control Allows Unrestricted Generation Incrementation

Severity	High
Finding ID	BH-2024-07-traitforge-04
Target	contracts/TraitForgeNft/TraitForgeNft.sol
Function name	_incrementGeneration

Description

The function `_incrementGeneration` is marked `private`, but if it is invoked from any public or external function without proper access control, it allows uncontrolled advancement of `currentGeneration`. There is no verification of whether the caller is authorized (e.g., onlyOwner or onlyAdmin), nor is there a hard cap on the number of generations. This could allow any user (if called indirectly) or even a misconfigured automation to prematurely or repeatedly increment generations, disrupting pricing logic and minting phases.

```
function _incrementGeneration() private {
    require(
      generationMintCounts[currentGeneration] >= maxTokensPerGen,
      'Generation limit not yet reached'
    );
    currentGeneration++;
    generationMintCounts[currentGeneration] = 0;
    priceIncrement = priceIncrement + priceIncrementByGen;
    entropyGenerator.initializeAlphaIndices();
    emit GenerationIncremented(currentGeneration);
  }
```

Recommendation

Ensure `_incrementGeneration` is only callable from functions that enforce strict access control (e.g., `onlyOwner`). Additionally, consider adding a cap (e.g., `maxGenerations`) and a guard clause like `require(currentGeneration < maxGenerations)` to prevent unbounded growth or logical bugs.

5. Funds Can Be Locked Due to Untrusted Ether Transfer in `nuke` Function

Severity	Medium
Finding ID	BH-2024-07-traitforge-05
Target	contracts/NukeFund/NukeFund.sol
Function name	nuke

Description

The `nuke` function performs a low-level Ether transfer to `msg.sender` using `.call{ value: claimAmount }('')`. If the recipient address is a smart contract with a fallback or receive function that reverts or consumes too much gas, the transfer will fail. Since the transaction includes `require(success, 'Failed to send Ether')`, this will cause the entire call to revert. As a result, the user cannot claim their funds, but the fund balance still retains the amount, making it effectively inaccessible — leading to a permanent lock of part of the fund. This introduces a denial-of-service vector: malicious users can intentionally mint or acquire NFTs and block fund withdrawal by using contracts that reject Ether.

```
function nuke(uint256 tokenId) public whenNotPaused nonReentrant {
    require(
      nftContract.isApprovedOrOwner(msg.sender, tokenId),
      'ERC721: caller is not token owner or approved'
    );
    require(
      nftContract.getApproved(tokenId) == address(this) ||
        nftContract.isApprovedForAll(msg.sender, address(this)),
      'Contract must be approved to transfer the NFT.'
    );
    require(canTokenBeNuked(tokenId), 'Token is not mature yet');

    uint256 finalNukeFactor =
calculateNukeFactor(tokenId); // finalNukeFactor has 5 digits
    uint256 potentialClaimAmount = (fund * finalNukeFactor) / MAX_DENOMINATOR;
    uint256 maxAllowedClaimAmount = fund / maxAllowedClaimDivisor;

    uint256 claimAmount = finalNukeFactor > nukeFactorMaxParam
      ? maxAllowedClaimAmount
      : potentialClaimAmount;

    fund -= claimAmount;

    nftContract.burn(tokenId);
    (bool success, ) = payable(msg.sender).call{ value: claimAmount }('');
    require(success, 'Failed to send Ether');

    emit Nuked(msg.sender, tokenId, claimAmount);
    emit FundBalanceUpdated(fund);
  }
```

Recommendation

Implement the pull-payment pattern: record `claimAmount` in a mapping (e.g., `claimable[msg.sender]`) and allow users to withdraw it via a separate `withdraw()` function. This decouples fund distribution from core logic, avoids reentrancy and DoS risks, and improves reliability. Additionally, validate `msg.sender` is not a contract or implement fallback protections if direct transfers must be used.

6. Lack of Slippage Protection in `mintWithBudget` May Lead to Fewer Mints Than Expected

Severity	Medium
Finding ID	BH-2024-07-traitforge-06
Target	contracts/TraitForgeNft/TraitForgeNft.sol
Function name	mintWithBudget

Description

The `mintWithBudget` function allows users to send a large `msg.value` and attempts to mint as many tokens as their budget allows. However, the mint price increases dynamically with each mint due to the logic in `calculateMintPrice`, which depends on `generationMintCounts[currentGeneration]`. This creates a race condition where the price can change between the time the user signs the transaction and when it is mined, leading to users minting fewer tokens than expected or being refunded unexpected amounts. This creates a poor user experience and potential economic loss, especially in automated systems or scripts expecting deterministic outcomes.

```
function mintWithBudget(
    bytes32[] calldata proof
  )
    public
    payable
    whenNotPaused
    nonReentrant
    onlyWhitelisted(proof, keccak256(abi.encodePacked(msg.sender)))
  {
    uint256 mintPrice = calculateMintPrice();
    uint256 amountMinted = 0;
    uint256 budgetLeft = msg.value;

    while (budgetLeft >= mintPrice && _tokenIds < maxTokensPerGen) {
      _mintInternal(msg.sender, mintPrice);
      amountMinted++;
      budgetLeft -= mintPrice;
      mintPrice = calculateMintPrice();
    }
    if (budgetLeft > 0) {
      (bool refundSuccess, ) = msg.sender.call{ value: budgetLeft }('');
      require(refundSuccess, 'Refund failed.');
    }
  }
```

Recommendation

Introduce slippage protection by allowing users to specify `minExpectedMints` or `maxAcceptablePricePerToken` as input parameters. Before minting, validate that the final mint price does not exceed expectations, and revert if slippage exceeds tolerance. Alternatively, allow users to precompute and lock in mint price via a quote mechanism, similar to how DEXes protect against price movement.

7. Unsafe and Ineffective Pseudo-Randomness in `writeEntropyBatch1`

Severity	Medium
Finding ID	BH-2024-07-traitforge-07
Target	contracts/EntropyGenerator/EntropyGenerator.sol
Function name	writeEntropyBatch1

Description

The `writeEntropyBatch1` function attempts to generate pseudo-random values using `keccak256(abi.encodePacked(block.number, i))`. However, `block.number` is fully predictable within the current block and `i` is a loop index, making the resulting entropy easily predictable and manipulatable by miners or observers. This undermines the purpose of generating entropy, especially if it is later used for trait assignment, airdrops, or any logic that relies on unpredictability. Additionally, the `require(pseudoRandomValue != 999999, 'Invalid value, retry.')` line introduces a hardcoded magic value rejection without a proper retry mechanism — if this condition fails during execution, the entire transaction will revert, making it unreliable and potentially DoS-prone.

```
function writeEntropyBatch1() public {
    require(lastInitializedIndex < batchSize1, 'Batch 1 already initialized.');

    uint256 endIndex = lastInitializedIndex + batchSize1;
    unchecked {
      for (uint256 i = lastInitializedIndex; i < endIndex; i++) {
        uint256 pseudoRandomValue = uint256(
          keccak256(abi.encodePacked(block.number, i))
        ) % uint256(10) ** 78;
        require(pseudoRandomValue != 999999, 'Invalid value, retry.');
        entropySlots[i] = pseudoRandomValue;
      }
    }
    lastInitializedIndex = endIndex;
  }
```

Recommendation

Do not rely on predictable inputs like `block.number` or loop indices for entropy generation. Instead, integrate a secure randomness oracle like Chainlink VRF or use a commit-reveal mechanism. If some values are considered invalid (e.g., `999999`), a robust retry or fallback mechanism should be implemented instead of reverting the entire batch. Also, verify that `endIndex` does not exceed the size of `entropySlots` to avoid out-of-bounds errors.

8. DAO Stage in `receive()` Function May Be Unreachable Due to Uncontrollable External State

Severity	Medium
Finding ID	BH-2024-07-traitforge-08
Target	contracts/NukeFund/NukeFund.sol
Function name	receive

Description

The logic in the `receive()` function sends a portion of the received ETH (`devShare`) to one of three addresses depending on the external state of `airdropContract` . Specifically, it checks `airdropStarted()` and `daoFundAllowed()` to determine whether to send funds to `devAddress` , `owner()` , or `daoAddress` . However, if the conditions for `daoFundAllowed()` are never met — due to misconfiguration, delay, or intentional behavior of the `airdropContract` — the branch for sending to `daoAddress` will never be executed. This makes the DAO stage logically unreachable, potentially violating governance expectations and permanently diverting funds away from the DAO. This breaks system assumptions around decentralization, protocol revenue sharing, or post-airdrop phases.

```
receive() external payable {
    uint256 devShare = msg.value / taxCut; // Calculate developer's share (10%)
    uint256 remainingFund = msg.value - devShare; // Calculate remaining funds to
add to the fund

    fund += remainingFund; // Update the fund balance

    if (!airdropContract.airdropStarted()) {
      (bool success, ) = devAddress.call{ value: devShare }('');
      require(success, 'ETH send failed');
      emit DevShareDistributed(devShare);
    } else if (!airdropContract.daoFundAllowed()) {
      (bool success, ) = payable(owner()).call{ value: devShare }('');
      require(success, 'ETH send failed');
    } else {
      (bool success, ) = daoAddress.call{ value: devShare }('');
      require(success, 'ETH send failed');
      emit DevShareDistributed(devShare);
    }

    emit FundReceived(msg.sender, msg.value); // Log the received funds
    emit FundBalanceUpdated(fund); // Update the fund balance
  }
```

Recommendation

Redesign the condition checks to ensure the DAO stage is reachable under well-defined, verifiable conditions. Consider adding a direct override or admin-controlled switch that explicitly transitions between fund distribution phases (e.g., from 'airdrop' to 'dao'). Emit events when such transitions occur, and document them clearly. Alternatively, use time-based or block-based constraints to prevent indefinite stalling at earlier stages.

9. Hardcoded Special Return Value Creates Hidden Control Flow and Reduces Entropy Quality

Severity	Medium
Finding ID	BH-2024-07-traitforge-09
Target	contracts/EntropyGenerator/EntropyGenerator.sol
Function name	getEntropy

Description

The `getEntropy` function includes a special case where it returns the hardcoded value `999999` if the requested slot and number index match a specific 'selection point' (`slotIndexSelectionPoint` and `numberIndexSelectionPoint`). This introduces an implicit control flow branch that is not obvious to users of this function. The value `999999` is not random or derived from entropy, but instead appears to serve as a sentinel or marker. This breaks the uniformity and unpredictability expected from an entropy function. If downstream logic treats `999999` differently (e.g., skips minting, reverts, or alters outcomes), then this behavior can be manipulated by users who understand the condition, leading to fairness or integrity issues. Additionally, there's no validation to ensure that the entropySlots[slotIndex] is initialized or non-zero, which could result in returning meaningless or padded zero entropy.

```
function getEntropy(
    uint256 slotIndex,
    uint256 numberIndex
  ) private view returns (uint256) {
    require(slotIndex <= maxSlotIndex, 'Slot index out of bounds.');

    if (
      slotIndex == slotIndexSelectionPoint &&
      numberIndex == numberIndexSelectionPoint
    ) {
      return 999999;
    }

    uint256 position = numberIndex * 6;
    require(position <= 72, 'Position calculation error');

    uint256 slotValue = entropySlots[slotIndex];
    uint256 entropy = (slotValue / (10 ** (72 - position))) % 1000000;
    uint256 paddedEntropy = entropy * (10 ** (6 - numberOfDigits(entropy)));

    return paddedEntropy;
  }
```

Recommendation

Avoid embedding sentinel values like `999999` in an entropy generation function. If a special condition must be flagged, use a separate return flag or structured data (e.g., `(bool valid, uint256 entropy)`), or emit an event. Also ensure that `entropySlots[slotIndex]` is checked for initialization before use, and that the entropy returned is cryptographically unpredictable and consistently distributed.

10. Division by Zero Risk and Unpredictable Age Due to Performance Factor from Entropy

Severity	Medium
Finding ID	BH-2024-07-traitforge-10
Target	contracts/NukeFund/NukeFund.sol
Function name	calculateAge

Description

The `calculateAge` function derives `perfomanceFactor` from `nftContract.getTokenEntropy(tokenId) % 10`, which means the factor can be zero. If `perfomanceFactor == 0`, then the resulting `age` will always be zero, regardless of how long the token has existed. This creates a silent logic flaw that may cause some tokens to never mature or qualify for downstream eligibility (e.g., nuking or airdrop rewards), especially if age is used as a maturity metric. This is particularly problematic because entropy is pseudo-random and not under user control, making it unpredictable which tokens will be affected. Additionally, the expression uses multiple chained divisions (e.g., `/60/60/24`) without precomputing, which may cause unintended truncation due to integer division.

```
function calculateAge(uint256 tokenId) public view returns (uint256) {
    require(nftContract.ownerOf(tokenId) != address(0), 'Token does not exist');

    uint256 daysOld = (block.timestamp -
      nftContract.getTokenCreationTimestamp(tokenId)) /
      60 /
      60 /
      24;
    uint256 perfomanceFactor = nftContract.getTokenEntropy(tokenId) % 10;

    uint256 age = (daysOld *
      perfomanceFactor *
      MAX_DENOMINATOR *
      ageMultiplier) / 365;
    return age;
  }
```

Recommendation

Ensure that `perfomanceFactor` is never zero. One solution is to modify the modulus to be `(entropy % 9) + 1`, which guarantees a range from 1 to 9. This avoids age being zero due to performance factor. Additionally, consider refactoring the timestamp division to reduce integer truncation errors and improve clarity. Document the age formula clearly for maintainability.

11. Forger Can Forge More Times Than Intended by Relisting Between Calls

Severity	Medium
Finding ID	BH-2024-07-traitforge-11
Target	contracts/EntityForging/EntityForging.sol
Function name	forgeWithListed

Description

The `forgeWithListed` function relies on `listings` to track the forge status and fee of the `forgerTokenId`. However, the forger's forge potential or forging limit is not rechecked during the `forgeWithListed` process. Instead, the comment suggests it is only enforced during the `listForForging` phase. As a result, a forger can list their token once (with their forge potential not yet maxed out), then forge multiple times with different merger tokens—exceeding their intended forge limit. This is possible because the function only increments `forgingCounts[forgerTokenId]` without checking if the count exceeds the forger's allowed limit. It assumes the forger can only be used once per listing, but nothing in the contract enforces this.

```
function forgeWithListed(
    uint256 forgerTokenId,
    uint256 mergerTokenId
  ) external payable whenNotPaused nonReentrant returns (uint256) {
    Listing memory _forgerListingInfo = listings[listedTokenIds[forgerTokenId]];
    require(
      _forgerListingInfo.isListed,
      "Forger's entity not listed for forging"
    );
    require(
      nftContract.ownerOf(mergerTokenId) == msg.sender,
      'Caller must own the merger token'
    );
    require(
      nftContract.ownerOf(forgerTokenId) != msg.sender,
      'Caller should be different from forger token owner'
    );
    require(
      nftContract.getTokenGeneration(mergerTokenId) ==
        nftContract.getTokenGeneration(forgerTokenId),
      'Invalid token generation'
    );

    uint256 forgingFee = _forgerListingInfo.fee;
    require(msg.value >= forgingFee, 'Insufficient fee for forging');

    _resetForgingCountIfNeeded(forgerTokenId);
    _resetForgingCountIfNeeded(mergerTokenId);

    forgingCounts[forgerTokenId]++;

    uint256 mergerEntropy = nftContract.getTokenEntropy(mergerTokenId);
    require(mergerEntropy % 3 != 0, 'Not merger');
    uint8 mergerForgePotential = uint8((mergerEntropy / 10) % 10);
    forgingCounts[mergerTokenId]++;
    require(
      mergerForgePotential > 0 &&
        forgingCounts[mergerTokenId] <= mergerForgePotential,
      'forgePotential insufficient'
    );

    uint256 devFee = forgingFee / taxCut;
    uint256 forgerShare = forgingFee - devFee;
    address payable forgerOwner = payable(nftContract.ownerOf(forgerTokenId));

    uint256 newTokenId = nftContract.forge(
      msg.sender,
      forgerTokenId,
      mergerTokenId,
      ''
    );
    (bool success, ) = nukeFundAddress.call{ value: devFee }('');
    require(success, 'Failed to send to NukeFund');
    (bool success_forge, ) = forgerOwner.call{ value: forgerShare }('');
    require(success_forge, 'Failed to send to Forge Owner');

    _cancelListingForForging(forgerTokenId);

    uint256 newEntropy = nftContract.getTokenEntropy(newTokenId);

    emit EntityForged(
      newTokenId,
      forgerTokenId,
      mergerTokenId,
      newEntropy,
      forgingFee
    );

    return newTokenId;
  }
```

Recommendation

Add a check during `forgeWithListed` to ensure that `forgingCounts[forgerTokenId]` does not exceed the token's forge potential (e.g., parsed from entropy). Relying solely on listing-time checks is insufficient since the forger can be reused by relisting. Also consider enforcing that a token cannot be re-listed if its forge potential is exhausted.

12. Lack of Slippage Protection in Token Nuking May Lead to Unexpected Claims

Severity	Medium
Finding ID	BH-2024-07-traitforge-12
Target	contracts/NukeFund/NukeFund.sol
Function name	nuke

Description

The `nuke` function calculates the `claimAmount` dynamically based on the current `fund` value and `finalNukeFactor`. However, there is no slippage check between when the user initiates the transaction and when the fund transfer occurs. If the fund is altered in the same block (e.g., via reentrancy or parallel transaction), the actual `claimAmount` may be significantly lower than expected. Additionally, users cannot specify a minimum acceptable claim amount to protect against unexpected value drops, potentially leading to economic loss or front-running vulnerabilities.

```
function nuke(uint256 tokenId) public whenNotPaused nonReentrant {
    require(
      nftContract.isApprovedOrOwner(msg.sender, tokenId),
      'ERC721: caller is not token owner or approved'
    );
    require(
      nftContract.getApproved(tokenId) == address(this) ||
        nftContract.isApprovedForAll(msg.sender, address(this)),
      'Contract must be approved to transfer the NFT.'
    );
    require(canTokenBeNuked(tokenId), 'Token is not mature yet');

    uint256 finalNukeFactor =
calculateNukeFactor(tokenId); // finalNukeFactor has 5 digits
    uint256 potentialClaimAmount = (fund * finalNukeFactor) / MAX_DENOMINATOR; //
Calculate the potential claim amount based on the finalNukeFactor
    uint256 maxAllowedClaimAmount = fund / maxAllowedClaimDivisor; // Define a
maximum allowed claim amount as 50% of the current fund size

    uint256 claimAmount = finalNukeFactor > nukeFactorMaxParam
      ? maxAllowedClaimAmount
      : potentialClaimAmount;

    fund -= claimAmount; // Deduct the claim amount from the fund

    nftContract.burn(tokenId); // Burn the token
    (bool success, ) = payable(msg.sender).call{ value: claimAmount }('');
    require(success, 'Failed to send Ether');

    emit Nuked(msg.sender, tokenId, claimAmount); // Emit the event with the actual
claim amount
    emit FundBalanceUpdated(fund); // Update the fund balance
  }
```

Recommendation

Introduce a `minClaimAmount` parameter in the `nuke` function so that the user can define the minimum acceptable value they expect to receive. Include a check like `require(claimAmount >= minClaimAmount, 'Slippage too high');`. This is standard practice for value-sensitive operations. Additionally, consider snapshotting fund state before processing if multiple claims may happen in a short window.

13. Excess ETH Sent to `forgeWithListed` May Become Permanently Stuck

Severity	Medium
Finding ID	BH-2024-07-traitforge-13
Target	contracts/EntityForging/EntityForging.sol
Function name	forgeWithListed

Description

The `forgeWithListed` function does not refund any excess ETH sent above the required `forgingFee`. If a user mistakenly sends more than the required `msg.value`, the contract deducts the exact `forgingFee` but does not return the remainder. As a result, any surplus ETH remains stuck in the contract unless manually withdrawn (which isn't possible if no withdraw function exists). Over time, this can accumulate to a meaningful amount, especially if users repeatedly overpay.

```
function forgeWithListed(
    uint256 forgerTokenId,
    uint256 mergerTokenId
  ) external payable whenNotPaused nonReentrant returns (uint256) {
    Listing memory _forgerListingInfo = listings[listedTokenIds[forgerTokenId]];
    require(
      _forgerListingInfo.isListed,
      "Forger's entity not listed for forging"
    );
    require(
      nftContract.ownerOf(mergerTokenId) == msg.sender,
      'Caller must own the merger token'
    );
    require(
      nftContract.ownerOf(forgerTokenId) != msg.sender,
      'Caller should be different from forger token owner'
    );
    require(
      nftContract.getTokenGeneration(mergerTokenId) ==
        nftContract.getTokenGeneration(forgerTokenId),
      'Invalid token generation'
    );
```

```
uint256 forgingFee = _forgerListingInfo.fee;
require(msg.value >= forgingFee, 'Insufficient fee for forging');

_resetForgingCountIfNeeded(forgerTokenId); // Reset for forger if needed
_resetForgingCountIfNeeded(mergerTokenId); // Reset for merger if needed

forgingCounts[forgerTokenId]++;

uint256 mergerEntropy = nftContract.getTokenEntropy(mergerTokenId);
require(mergerEntropy % 3 != 0, 'Not merger');
uint8 mergerForgePotential = uint8((mergerEntropy / 10) % 10);
forgingCounts[mergerTokenId]++;
require(
  mergerForgePotential > 0 &&
    forgingCounts[mergerTokenId] <= mergerForgePotential,
  'forgePotential insufficient'
);

uint256 devFee = forgingFee / taxCut;
uint256 forgerShare = forgingFee - devFee;
address payable forgerOwner = payable(nftContract.ownerOf(forgerTokenId));

uint256 newTokenId = nftContract.forge(
  msg.sender,
  forgerTokenId,
  mergerTokenId,
  ''
);
(bool success, ) = nukeFundAddress.call{ value: devFee }('');
require(success, 'Failed to send to NukeFund');
(bool success_forge, ) = forgerOwner.call{ value: forgerShare }('');
require(success_forge, 'Failed to send to Forge Owner');

_cancelListingForForging(forgerTokenId);

uint256 newEntropy = nftContract.getTokenEntropy(newTokenId);

emit EntityForged(
  newTokenId,
  forgerTokenId,
  mergerTokenId,
  newEntropy,
  forgingFee
```

```
    );

    return newTokenId;
  }
```

Recommendation

Refund the excess ETH after the `forgingFee` is split and transferred. Add the following logic at the end of the function:

```
uint256 refund = msg.value - forgingFee;
if (refund > 0) {
  (bool refunded, ) = payable(msg.sender).call{value: refund}("");
  require(refunded, "Refund failed");
}
```

Alternatively, require exact payment via `require(msg.value == forgingFee)` if overpayment is undesired.

14. Unused Variable `amountMinted` in `mintWithBudget` Function

Severity	Low
Finding ID	BH-2024-07-traitforge-14
Target	contracts/TraitForgeNft/TraitForgeNft.sol
Function name	mintWithBudget

Description

The variable `amountMinted` is incremented in each iteration of the while loop to track the number of tokens minted within the given budget. However, it is never used after being calculated, making it a redundant piece of state. This adds unnecessary gas cost and may confuse readers or future maintainers of the contract.

```
function mintWithBudget(
    bytes32[] calldata proof
  )
    public
    payable
    whenNotPaused
    nonReentrant
    onlyWhitelisted(proof, keccak256(abi.encodePacked(msg.sender)))
  {
    uint256 mintPrice = calculateMintPrice();
    uint256 amountMinted = 0;
    uint256 budgetLeft = msg.value;

    while (budgetLeft >= mintPrice && _tokenIds < maxTokensPerGen) {
      _mintInternal(msg.sender, mintPrice);
      amountMinted++;
      budgetLeft -= mintPrice;
      mintPrice = calculateMintPrice();
    }
    if (budgetLeft > 0) {
      (bool refundSuccess, ) = msg.sender.call{ value: budgetLeft }('');
      require(refundSuccess, 'Refund failed.');
    }
  }
```

Recommendation

Remove the `amountMinted` variable if it is not intended to be used. If tracking minted amounts is needed for logging or logic, consider emitting an event or using it in a return value.

```
// Remove this line:
uint256 amountMinted = 0;

// And this line:
amountMinted++;
```

15. Redundant Condition Can Be Simplified in `_mintNewEntity`

Severity	Low
Finding ID	BH-2024-07-traitforge-15
Target	contracts/TraitForgeNft/TraitForgeNft.sol
Function name	_mintNewEntity

Description

The condition `generationMintCounts[gen] >= maxTokensPerGen && gen == currentGeneration` checks whether the current generation has reached its minting limit. However, the `require` statement earlier in the function already ensures that `generationMintCounts[gen] < maxTokensPerGen`, so the `>=` comparison will always be true immediately after the increment `generationMintCounts[gen]++`, if it reaches the limit. Therefore, this logic is correct, but slightly redundant in expression and can be made more readable by inlining or reordering operations.

```
function _mintNewEntity(
    address newOwner,
    uint256 entropy,
    uint256 gen
  ) private returns (uint256) {
    require(
      generationMintCounts[gen] < maxTokensPerGen,
      'Exceeds maxTokensPerGen'
    );

    _tokenIds++;
    uint256 newTokenId = _tokenIds;
    _mint(newOwner, newTokenId);

    tokenCreationTimestamps[newTokenId] = block.timestamp;
    tokenEntropy[newTokenId] = entropy;
    tokenGenerations[newTokenId] = gen;
    generationMintCounts[gen]++;
    initialOwners[newTokenId] = newOwner;

    if (
      generationMintCounts[gen] >= maxTokensPerGen && gen == currentGeneration
    ) {
      _incrementGeneration();
    }

    if (!airdropContract.airdropStarted()) {
      airdropContract.addUserAmount(newOwner, entropy);
    }

    emit NewEntityMinted(newOwner, newTokenId, gen, entropy);
    return newTokenId;
  }
```

Recommendation

You can simplify or clarify the condition after increment:

```
++generationMintCounts[gen];
if (generationMintCounts[gen] == maxTokensPerGen && gen == currentGeneration) {
  _incrementGeneration();
}
```

This removes the confusion between `<` and `>=` and clearly expresses the intent: "after this mint, if we've just hit the cap for the current generation, increment it."

16. fetchListings Returns Array with Unused Zero Index and Possible Empty Entries

Severity	Low
Finding ID	BH-2024-07-traitforge-16
Target	contracts/EntityForging/EntityForging.sol
Function name	fetchListings

Description

The `fetchListings` function initializes an array of size `listingCount + 1`, but skips index 0 and begins populating from index 1. As a result, the returned array contains an unused (default zero-value) element at index 0. This can be inefficient and confusing for consumers of the function, especially if they expect listings to be packed from index 0. Additionally, it may return entries for inactive or deleted listings, depending on how `listings[i]` is managed.

```
function fetchListings() external view returns (Listing[] memory _listings) {
    _listings = new Listing[](listingCount + 1);
    for (uint256 i = 1; i <= listingCount; ++i) {
      _listings[i] = listings[i];
    }
  }
```

Recommendation

Refactor the function to return a compact array of active listings, starting from index 0. Consider filtering only active ones using an additional count or temporary dynamic array, then copying into a fixed-size array. Alternatively, document clearly that index 0 is unused, if that pattern is intentional.

17. Seller Can Block NFT Sale by Reverting ETH Transfer

Severity	Low
Finding ID	BH-2024-07-traitforge-17
Target	contracts/EntityTrading/EntityTrading.sol
Function name	buyNFT

Description

In the `buyNFT` function, the seller is paid via a low-level `call` to `listing.seller`. If the seller is a contract with a fallback or receive function that deliberately reverts the transaction, the ETH transfer will fail, and the entire purchase will revert. This enables a malicious or misconfigured seller to intentionally block any sale, causing a denial of service for potential buyers.

```
function buyNFT(uint256 tokenId) external payable whenNotPaused nonReentrant {
    Listing memory listing = listings[listedTokenIds[tokenId]];
    require(
      msg.value == listing.price,
      'ETH sent does not match the listing price'
    );
    require(listing.seller != address(0), 'NFT is not listed for sale.');

    //transfer eth to seller (distribute to nukefund)
    uint256 nukeFundContribution = msg.value / taxCut;
    uint256 sellerProceeds = msg.value - nukeFundContribution;
    transferToNukeFund(nukeFundContribution); // transfer contribution to nukeFund

    // transfer NFT from contract to buyer
    (bool success, ) = payable(listing.seller).call{ value: sellerProceeds }(
      ''
    );
    require(success, 'Failed to send to seller');
    nftContract.transferFrom(address(this), msg.sender,
tokenId); // transfer NFT to the buyer

    delete listings[listedTokenIds[tokenId]]; // remove listing

    emit NFTSold(
      tokenId,
      listing.seller,
      msg.sender,
      msg.value,
      nukeFundContribution
    ); // emit an event for the sale
  }
```

Recommendation

Use the pull payment pattern or OpenZeppelin’s `PaymentSplitter` to allow sellers to withdraw funds explicitly. Alternatively, ensure seller contracts are safe to receive ETH or use safer transfer mechanisms with fallback recovery options.

18. Ambiguous Modulus Usage in Entropy Generation

Severity	Low
Finding ID	BH-2024-07-traitforge-18
Target	contracts/EntropyGenerator/EntropyGenerator.sol
Function name	writeEntropyBatch2

Description

In the `writeEntropyBatch2` function, entropy values are generated using a modulus operation with a large base (`10 ** 78`). However, the same function checks if the resulting value is equal to `999999` , which is only 6 digits. This creates ambiguity, as it's unclear whether the intention was to constrain the value to a 6-digit space or operate in a much larger numeric space. This inconsistency could lead to unexpected or incorrect logic behavior and complicate validation and debugging efforts.

```
function writeEntropyBatch2() public {
    require(
      lastInitializedIndex >= batchSize1 && lastInitializedIndex < batchSize2,
      'Batch 2 not ready or already initialized.'
    );

    uint256 endIndex = lastInitializedIndex + batchSize1;
    unchecked {
      for (uint256 i = lastInitializedIndex; i < endIndex; i++) {
        uint256 pseudoRandomValue = uint256(
          keccak256(abi.encodePacked(block.number, i))
        ) % uint256(10) ** 78;
        require(pseudoRandomValue != 999999, 'Invalid value, retry.');
        entropySlots[i] = pseudoRandomValue;
      }
    }
    lastInitializedIndex = endIndex;
  }
```

Recommendation

Clarify the intended value range by aligning the modulus base with the condition (e.g., use `10 ** 6` if you're checking against `999999`). If a large entropy space is required, then revise the comparison to match the bit size or range correctly, and document the intent clearly.