code4rena

Traitforge

Smart Contract Security Assessment

Version 2.0

Audit dates: Oct 30 — Nov 04, 2024

Audited by: SpicyMeatball
VictorMagnum

Contents

1. Introduction

1.1 About Zenith

1.2 Disclaimer

1.3 Risk Classification

2. Executive Summary

2.1 About Traitforge

2.2 Scope

2.3 Audit Timeline

2.4 Issues Found

3. Findings Summary

4. Findings

4.1 Critical Risk

4.2 High Risk

4.3 Medium Risk

4.4 Low Risk

4.5 Informational

1. Introduction

1.1 About Zenith

Zenith is an offering by Code4rena that provides consultative audits from the very best security researchers in the space. We focus on crafting a tailored security team specifically for the needs of your codebase.

Learn more about us at https://code4rena.com/zenith.

1.2 Disclaimer

This report reflects an analysis conducted within a defined scope and time frame, based on provided materials and documentation. It does not encompass all possible vulnerabilities and should not be considered exhaustive.

The review and accompanying report are presented on an "as-is" and "as-available" basis, without any express or implied warranties.

Furthermore, this report neither endorses any specific project or team nor assures the complete security of the project.

1.3 Risk Classification

SEVERITY LEVEL	IMPACT: HIGH	IMPACT: MEDIUM	IMPACT: LOW
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

2. Executive Summary

2.1 About Traitforge

TraitForge is the dynamic new NFT-fi game where players Mint, Trade, Forge, and Nuke their NFT Entities in wildly variable strategies to win. By working with and competing against other Forgers, players can claim shares of the ever-changing $ETH Nuke Fund at the center of the game.

2.2 Scope

Repository	TraitForge/traitforge-contracts/
Commit Hash	dd4ec9fa0d020b9d4f958e9db8a66d4ed1be3c79
Mitigation Hash	b54f60bc5902587da052d794d36489f5d0d90601

2.3 Audit Timeline

DATE	EVENT
Oct 30, 2024	Audit start
Nov 04, 2024	Audit end
Nov 19, 2024	Report published

2.4 Issues Found

SEVERITY	COUNT
Critical Risk	1
High Risk	1
Medium Risk	3
Low Risk	2
Informational	2
Total Issues	9

3. Findings Summary

ID	DESCRIPTION	STATUS
C-1	Participants can brick the contract and prevent the winner from getting their rewards	Resolved

H-1	fulfillRandomWords can revert and stall the round	Resolved
M-1	A migration can be triggered while there's an active round	Resolved
M-2	Golden God token can bypass bid limit restrictions	Resolved
M-3	Collision can occur when NFTs are burned in `bidPayout`	Resolved
L-1	Anyone can return data bomb the bidPayout call	Resolved
L-2	Additional validation is recommended in the setters for `quantityToBeBurnt` and `numWords`	Resolved
I-1	Upper bound for callback gas limit	Resolved
I-2	`getMaxBidPotential` will return wrong value for Golden God NFT	Resolved

4. Findings

4.1 Critical Risk

A total of 1 critical risk findings were identified.

[C-1] Participants can brick the contract and prevent the winner from getting their rewards

Severity: Critical	Status: Resolved

Context:

- LottFund.sol

Description:

When users bid, we check if they have approved the contract:

```
        if (
            !(
                traitForgeNft.getApproved(tokenId) == address(this)
                    || traitForgeNft.isApprovedForAll(msg.sender,
address(this))
            )
```

However, after bidding they can revoke their approvals and this is quite problematic as it can brick contract. Let me explain how:

When requestRandomWords is called, the contract is *briefly paused*, meaning new randomness requests cannot be made (hence bids cannot be made too).

On the other hand, when `fulfillRandomWords` is called, it calls `bidPayout`.

Now, consider the following scenario:

1. User bids
2. User notices that they are not a winner, but their token will be burnt by monitoring the mempool w the fulfillRandomWords request.
3. User revokes their approval and now `burnTokens` cannot go through.
4. As a result `resetRound` cannot go through and the winner cannot get their reward.

Finally, no more rounds can be started, because there's no way to reset the round and go to the next one.

Recommendation: Consider holding the NFT's in the contract while the round is active

Client: Mitigated in @26ba66570acb..

Zenith: Verified.

4.2 High Risk

A total of 1 high risk findings were identified.

[H-1] fulfillRandomWords can revert and stall the round

Severity: High	Status: Resolved

Context:

- LottFund.sol

Description: fulfillRandomWords can revert and it shouldn't as per Chainlink VRF Security Considerations, because the VRF service will not attempt to call it a second time.

```
        if (bidsAmount != maxBidAmount) {
            //if bidsAmount has no maxxed out then revert
            revert LottFund__BiddingNotFinished();
        }
```

This can revert due to a bug in batchBid, where the loop doesn't break if bidsAmount >= maxBidAmount, see https://github.com/code-423n4/2024-10-traitforge-zenith/pull/1#discussion_r1821980516. Fixing it should resolve the issue.

```
              (bool success,) = payable(ownerOfWinningToken).call{ value:
claimAmount }("");
            require(success, "Failed to send Ether");
```

This can also revert if the winner's receive fallback function reverts the transaction. Although unlikely, if it occurs, it would stall the round.

Recommendation: Implementing a pull-based approach is recommended, allowing users to claim rewards instead of sending them in bidPayout.

Client: Mitigated in @26ba66570acb..

Zenith: Verified.

4.3 Medium Risk

A total of 3 medium risk findings were identified.

[M-1] A migration can be triggered while there's an active round

Severity: Medium	Status: Resolved

Context:

- LottFund.sol

Description:

Migration should only probably happen while pausedBids = true, cause doing it mid-round can have many unexpected consequences and lead to invalid state.

```
    function migrate(address newAddress) external whenNotPaused
onlyProtocolMaintainer {
        require(newAddress != address(0), "Invalid new contract
address");
        uint256 contractBalance = address(this).balance;
        if (contractBalance > 0) {
            (bool success,) = newAddress.call{ value: contractBalance }
("");
            require(success, "Failed to transfer ETH");
        }
    }
```

Recommendation:

Ensure that a migration will happen only with the whole contract paused

Client: Fixed in https://github.com/TraitForge/traitforge-contracts/commit/f45f1496feb4995697bd3742109d1cc000963151

Zenith: Verified

[M-2] Golden God token can bypass bid limit restrictions

Severity: Medium	Status: Resolved

Context:

- LottFund.sol#L199-L202

Description: Users participate in the lottery by bidding their TraitForge NFTs using the `bid` and `batchBid` functions:

```
    function bid(uint256 tokenId) public whenNotPaused nonReentrant {
        if (pausedBids) revert LottFund__BiddingIsPaused();
        if (bidCountPerRound[currentRound][msg.sender] >=
maxBidsPerAddress) {
            revert LottFund__AddressHasBiddedTooManyTimes(msg.sender);
        }
        ITraitForgeNft traitForgeNft = _getTraitForgeNft();
        if (traitForgeNft.ownerOf(tokenId) != msg.sender) revert
LottFund__CallerNotTokenOwner();
        if (
            !(
                traitForgeNft.getApproved(tokenId) == address(this)
                    || traitForgeNft.isApprovedForAll(msg.sender,
address(this))
            )
        ) revert LottFund__ContractNotApproved();
>>      if (!canTokenBeBidded(tokenId)) revert
LottFund__TokenCannotBeBidded();
        bidCountPerRound[currentRound][msg.sender]++;
        bidsAmount++; // increase total bid amounts as max is currently
1500 (can be altered)
        tokenIdsBidded.push(tokenId); // store the array of tokenIds that
have been bidded

        ---SNIP---
    }
```

A token's bidding limit is determined by its entropy and the `maxModulusForToken` value:

```
    function canTokenBeBidded(uint256 tokenId) public view returns (bool)
{
        ITraitForgeNft traitForgeNft = _getTraitForgeNft();
```

```
        uint256 entropy = traitForgeNft.getTokenEntropy(tokenId); // get
entities 6 digit entropy
            // 999999 =
>>      if (entropy == 999_999) {
            // if golden god (999999) maxBidPotential is +1 over max
            return true;
        }

>>      uint256 tokensMaxBidPotential = entropy % maxModulusForToken; //
calculation for maxBidPotenital from entropy eg
        if (tokensMaxBidPotential == 0) {
            // if tokens maxBidePotential is 0 revert
            return false;
        }
>>      if (tokensMaxBidPotential <= tokenBidCount[tokenId]) {
            // if tokens maxBidPotential is less than or equal to how
many times it has bidded before then revert
            return false; // eg if the tokens maxBidPotential is 2 and it
has bidded twice
                // then it cannot bid again
        }
        return true;
    }
```

However, the special case for a token with entropy = 999_999 (Golden God) allows it to bypass the `tokenBidCount` validation, theoretically permitting infinite bids instead of enforcing the intended `maxBidPotential + 1`.

Recommendation: The comment in the code hints that the Golden God token should be allowed to bid one additional time above its max limit. Adjust the `canTokenBeBidded` logic to enforce this intended limit.

Client: Fixed in the following commit

Zenith: Verified

[M-3] Collision can occur when NFTs are burned in `bidPayout`

Severity: Medium	Status: Resolved

Context:

- LottFund.sol#L392-L393
- LottFund.sol#L405

Description: At the end of the lottery, a winner is selected based on random numbers provided by Chainlink VRF, and some participants' tokens are randomly chosen to be burned:

```
    function bidPayout(uint256[] calldata _randomWords) internal
whenNotPaused nonReentrant {
        ---SNIP---

        uint256[] memory tokensToBurn = new uint256[](quantityToBeBurnt);
//memory to stre the tokens to be burnt
        for (uint256 i = 1; i <= quantityToBeBurnt; i++) {
            // Use the next 5 numbers to locate the indexes of 5 tokenIds
to burn
>>          uint256 burnIndex = _randomWords[i] % tokenIdsBidded.length;
// Find the burn index
            tokensToBurn[i - 1] = tokenIdsBidded[burnIndex]; // Store the
token ID to burn
        }
        burnTokens(tokensToBurn); // Burn the selected tokenIds
        resetRound();
```

Tokens are chosen based on the random number, the number of bid tokens, and their positions in the `tokenIdsBidded` array. There is a possibility that `_randomWords[i] % tokenIdsBidded.length` could yield the same result across multiple iterations.

For example, if `randomWords[1] = 1500` and `randomWords[2] = 3000`, then `randomWords[i] % 1500` would give an index of 0 for both cases. This would cause the contract to attempt burning the same token twice, causing the `fulfillRandomWords` transaction to revert. Since Chainlink does not retry transactions, the contract would remain stuck in the current round.

Another possible outcome is that `_randomWords[i] % tokenIdsBidded.length` could select the winning token's index, meaning the winning token would be burned, which is unintended.

While these outcomes are relatively unlikely with default values of `maxBidAmount = 1500` and `quantityToBeBurnt = 5`, the probability increases if the owner adjusts these values, such as by reducing `maxBidAmount` or increasing `quantityToBeBurnt`.

The ability for users to bid the same token multiple times also increases the likelihood of a collision.

Recommendation: If a duplicate or the winning token is selected, it is recommended that the next element in the array be selected. Alternatively, a simpler solution would be to modify the `burnTokens` function to skip duplicates and winning tokens during the burn process.

Client: Fixed in the following commit

Zenith: Verified. The burning of NFTs has been removed from the `bidPayout` function.

4.4 Low Risk

A total of 2 low risk findings were identified.

[L-1] Anyone can return data bomb the bidPayout call

Severity: Low	Status: Resolved

Context:

- LottFund.sol

Description: External calls can cause "return data bombing" where it returns so much data that all gas is exhausted in copying it to memory.

Recommendation:

```
(bool success,) = payable(ownerOfWinningToken).call{ value: claimAmount }
("");
```

You can use ExcessivelySafeCall where you can limit the size of data being copied to memory

Client: Mitigated in @26ba66570acbe6..

Zenith: Verified.

[L-2] Additional validation is recommended in the setters for `quantityToBeBurnt` and `numWords`

Severity: Low	Status: Resolved

Context:

- LottFund.sol#L231
- LottFund.sol#L279
- LottFund.sol#L392

Description: When the lottery concludes, the contract randomly selects several tokens to be burned:

```
        uint256[] memory tokensToBurn = new uint256[](quantityToBeBurnt);
//memory to stre the tokens to be burnt
        for (uint256 i = 1; i <= quantityToBeBurnt; i++) {
            // Use the next 5 numbers to locate the indexes of 5 tokenIds
to burn
            uint256 burnIndex = _randomWords[i] % tokenIdsBidded.length;
// Find the burn index
            tokensToBurn[i - 1] = tokenIdsBidded[burnIndex]; // Store the
token ID to burn
        }
```

In the loop, tokens are selected based on random numbers provided by Chainlink VRF. If `_randomWords` array length is less than `quantityToBeBurnt + 1`, the transaction will fail due to an array out-of-bounds error.

Recommendation: Consider adding validation in `setNumWords` and `setAmountToBeBurnt` to ensure that `numWords >= quantityToBeBurnt + 1`.

Client: Fixed in the following commit

Zenith: Verified.

4.5 Informational

A total of 2 informational findings were identified.

[I-1] Upper bound for callback gas limit

Severity: Informational	Status: Resolved

Context:

- LottFund.sol

Description:

```
    function setCallbackGasLimit(uint32 _limit) external
onlyProtocolMaintainer {
        callbackGasLimit = _limit;
```

Max callback limit for Op/Base is `2500000`, perhaps a good sanity check to be added here. Or you can dynamically check it by getting it from the VRF coordinator directly.

https://basescan.org/address/0xd5D517aBE5cF79B7e95eC98dB0f0277788aFF634#readContract

That would be ideal.

Client: Fixed in @5dffd9808e48..

Zenith: Verified

[I-2] `getMaxBidPotential` will return wrong value for Golden God NFT

Severity: Informational	Status: Resolved

Context:

- LottFund.sol#L315-L320

Description: The `getMaxBidPotential` function calculates the maximum number of times an NFT can be bid on based on its entropy:

```
    function getMaxBidPotential(uint256 tokenId) public view returns
(uint256) {
        ITraitForgeNft traitForgeNft = _getTraitForgeNft();
        uint256 entropy = traitForgeNft.getTokenEntropy(tokenId);
        uint256 tokenMaxBidPotential = entropy % maxModulusForToken;
        return tokenMaxBidPotential;
    }
```

However, there is a special case: if a token has an entropy of 999_999 (Golden God), it should have an additional bid allowance beyond its calculated max bid potential. Currently, this exception is not accounted for in `getMaxBidPotential`.

Recommendation: Consider adding a condition to handle the Golden God token so that it reflects the additional bid potential in the `getMaxBidPotential` function.

Client: Fixed in the following commit

Zenith: Verified