shieldify

Bullas

SECURITY REVIEW

Date: 27 March 2025

CONTENTS

1. About Shieldify

Positioned as the first hybrid Web3 Security company, Shieldify shakes things up with a unique subscription-based auditing model that entitles the customer to unlimited audits within its duration, as well as top-notch service quality thanks to a disruptive 6-layered security approach. The company works with very well-established researchers in the space and has secured multiple millions in TVL across protocols, also can audit codebases written in Solidity, Vyper, Rust, Cairo, Move and Go.

Learn more about us at shieldify.org.

2. Disclaimer

This security review does not guarantee bulletproof protection against a hack or exploit. Smart contracts are a novel technological feat with many known and unknown risks. The protocol, which this report is intended for, indemnifies Shieldify Security against any responsibility for any misbehavior, bugs, or exploits affecting the audited code during any part of the project's life cycle. It is also pivotal to acknowledge that modifications made to the audited code, including fixes for the issues described in this report, may introduce new problems and necessitate additional auditing.

3. About Bullas

BULL-ISH is the first game in a new ecosystem of onchain games and experimental apps that utilize Beradrome to coordinate incentives, bringing greater depth to the concept of game points and assets that connect to the wider Berachain ecosystem.

When playing Bull Ish, you'll do two main things:

1. Spank a Bera Pay a fee in BERA to spank.

Spanking earns you Moola and, more importantly, puts you on a queue (breadline) that yields oBERO as long as you remain on it.

The breadline has a set capacity; once new spankers arrive, you're pushed along and eventually off the line, ending your `oBERO` earnings.

2. Build your Weapons Inventory Upgrading your Weapons boosts Moola production and increases your spank power.

This allows you to earn moola passively without spanking and earn more when you do spank.

4. Risk Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

4.1 Impact

- **High** – results in a significant risk for the protocol's overall well-being. Affects all or most users

- **Medium** – results in a non-critical risk for the protocol affects all or only a subset of users, but is still unacceptable
- **Low** – losses will be limited but bearable – and covers vectors similar to griefing attacks that can be easily repaired

4.2 Likelihood

- **High** – almost certain to happen and highly lucrative for execution by malicious actors
- **Medium** – still relatively likely, although only conditionally possible
- **Low** – requires a unique set of circumstances and poses non-lucrative cost-of-execution to rewards ratio for the actor

5. Security Review Summary

The security review lasted 2.5 days with a total of 40 hours dedicated by 2 researchers from the Shieldify team.

Overall, the code is well-written. The audit report identified one high-severity issue, two medium-severity issues, and one low-severity vulnerability. The vulnerabilities primarily stem from overpayment due to price reductions, missing transfer hooks for claiming rewards, and incorrect token distribution calculations.

The Bullas team has done a great job with the development and has been highly responsive to the Shieldify research team's inquiries and promptly implemented all recommendations.

5.1 Protocol Summary

Project Name	**Bullas**
Repository	bull-ish
Type of Project	DeFi, Gaming
Audit Timeline	2.5 days
Review Commit Hash	3b48a94280e1ddcc25f486735508a50c70a21741
Fixes Review Commit Hash	806a26913363d020c99636f7d87c1155efad2531

5.2 Scope

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

File	nSLOC
contracts/Factory.sol	122
contracts/Moola.sol	26
contracts/Multicall.sol	112
contracts/QueuePlugin.sol	245
Total	**505**

6. Findings Summary

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

- **Critical/High** issues: **1**
- **Medium** issues: **1**
- **Low** issues: **1**
- **Info** issues: **1**

ID	Title	Severity	Status
[H-01]	Excess Payment When Plugin Owner Reduces Price	High	Fixed
[M-01]	Missing Transfer Hook for Claiming Continuously Accruing Rewards	Medium	Acknowledged
[L-01]	Incorrect Token Amount Calculation in Distribution Logic	Low	Fixed
[I-01]	Missing Transfer Hook for Claiming Continuously Accruing Rewards	Info	Fixed

7. Findings

[H-01] Excess Payment When Plugin Owner Reduces Price

Severity

High Risk

Description

The `click()` function in the `Multicall` contract does not verify that the received `msg.value` matches the calculated `price`, which is determined by calling `IQueuePlugin(plugin).getPrice() * amount`. This creates a vulnerability where users may send excessive ETH that gets locked in the contract when the plugin owner decreases the `entryFee`.

```
function click(uint256 tokenId, uint256 amount, string calldata message)
    external payable returns (uint256 mintAmount) {
uint256 price = IQueuePlugin(plugin).getPrice() * amount;
IWBERA(base).deposit{value: price}();
// No validation that msg.value == price
IERC20(base).safeApprove(plugin, 0);
IERC20(base).safeApprove(plugin, price);

for (uint256 i = 0; i < amount; i++) {
    mintAmount += IQueuePlugin(plugin).click(tokenId, message);
}
}
```

The plugin owner can modify the `entryFee` through the `setEntryFee()` function, which directly impacts the price calculation:

```
function getPrice() external view returns (uint256) {
    return entryFee;
}
```

```
function setEntryFee(uint256 _entryFee) external onlyOwner {
    entryFee = _entryFee;
    emit Plugin__EntryFeeSet(_entryFee);
}
```

Location of Affected Code

File: Multicall.sol

Impact

Users may send excessive ETH that gets locked in the contract when the plugin owner decreases the `entryFee`.

Proof of Concept

1. A user calculates the required payment based on the current `entryFee` (e.g., 1 ETH per click) and amount (e.g., 10 clicks), resulting in 10 ETH.
2. The user sends a transaction with 10 ETH as `msg.value` to call the `click` function.
3. Before the user's transaction is processed, the plugin owner reduces the `entryFee` from 1 ETH to 0.5 ETH.
4. When the user's transaction executes, the function calculates `price = 0.5 ETH * 10 = 5 ETH`.
5. Only 5 ETH is used in the `deposit` call, while the remaining 5 ETH is stuck in the contract with no mechanism to retrieve it.

Recommendation

Check the relationship between `msg.value` and `price`. If there's extra, return it to the user; if it's not enough, revert.

Team Response

Fixed.

[M-01] Missing Transfer Hook for Claiming Continuously Accruing Rewards

Severity

Medium Risk

Description

The `Factory` contract has an issue with its reward-claiming mechanism. The contract continuously accrues rewards for NFT holders on a per-block basis but lacks a mechanism to automatically claim these rewards before ownership changes.

```
function claim(uint256 tokenId) public tokenExists(tokenId) {
    uint256 amount = tokenId_Ups[tokenId] * (block.timestamp -
       tokenId_Last[tokenId]);
    uint256 maxAmount = tokenId_Ups[tokenId] * DURATION;
    if (amount > maxAmount) amount = maxAmount;
    tokenId_Last[tokenId] = block.timestamp;
    emit Factory__Claimed(tokenId, amount);
    IUnits(units).mint(IERC721(key).ownerOf(tokenId), amount);
}
```

```
contract Bullas is ERC721Enumerable {

    uint256 public currentTokenId;

    constructor() ERC721("Bullas", "BULLAS") {}

    function mint() external returns (uint256) {
        uint256 newTokenId = ++currentTokenId;
        _mint(msg.sender, newTokenId);
        return newTokenId;
    }
}
```

The issue is problematic because: 1. Rewards accrue continuously with each new block 2. To claim all earned rewards before transferring an NFT, users must execute both operations in the same block 3. The claim transaction must occur before the transfer transaction within that block

This sequence requires technical knowledge beyond what typical users possess, as it involves understanding block timing, transaction ordering, and potentially using specialized tools to ensure transactions are processed in the correct order within a single block.

Location of Affected Code

File: Bullas.sol

File: Factory.sol#L76-L83

Impact

It's hard for users to claim all their rewards before the transfer, and they might lose some rewards. Check out the PoC below for a specific example.

Proof of Concept

1. Alice has NFT #123 and accrues 10 units of reward per block
2. At block 100, Alice decides to transfer the NFT to Bob
3. For Alice to receive all rewards accrued up to the transfer, she must:

- Submit a claim transaction
- Submit a transfer transaction
- Ensure both transactions are included in the same block (block 101)
- Ensure the claim transaction is processed before the transfer transaction

4. If Alice's claim transaction is processed in block 101 but the transfer occurs in block 102, she loses rewards accrued during block 101
5. If both transactions are in block 101 but the transfer executes before the claim, the rewards are minted to Bob instead of Alice
6. If Alice simply transfers without coordinating these complex requirements, she loses all accumulated rewards

Recommendation

The contract should implement a transfer hook mechanism that automatically calls `claim()` when an NFT is transferred, ensuring that the original owner receives all accrued rewards up to the moment of transfer without requiring complex transaction orchestration.

Team Response

Acknowledged.

[L-01] Incorrect Token Amount Calculation in Distribution Logic

Severity

Low Risk

Description

There is a precision issue in the `claimAndDistribute()` function of the `QueuePlugin` contract. The function calculates a treasury fee and distributes it between treasury and developer addresses, but when determining the remaining amount for bribes or additional treasury transfer, it uses the calculated `treasuryFee` value rather than tracking the actual amounts transferred.

```
function claimAndDistribute()
    external
    nonReentrant
{
    uint256 balance = token.balanceOf(address(this));
    if (balance > DURATION) {
        uint256 treasuryFee = balance / 5;
        token.safeTransfer(treasury, treasuryFee * 3 / 5);
        token.safeTransfer(developer, treasuryFee * 2 / 5);
        if (autoBribe) {
            token.safeApprove(bribe, 0);
            token.safeApprove(bribe, balance - treasuryFee);
            IBribe(bribe).notifyRewardAmount(address(token), balance -
               treasuryFee);
        } else {
            token.safeTransfer(treasury, balance - treasuryFee);
        }
        emit Plugin__ClaimedAndDistributed(balance);
    }
}
```

Due to integer division, the sum of `treasuryFee * 3 / 5` and `treasuryFee * 2 / 5` may be less than the original `treasuryFee` due to rounding down. This will result in the dust caused by round down not being handled, so the actual amount processed is less than it should be.

Location of Affected Code

File: QueuePlugin.sol#L166-L184

Impact

This will result in the dust caused by round down not being handled, so the actual amount processed is less than it should be.

Proof of Concept

1. The contract has a balance of 30 tokens
2. `treasuryFee = 30 / 5 = 6` tokens
3. Treasury receives `6 * 3 / 5 = 3` tokens (integer division rounds down from 3.6)
4. The developer receives `6 * 2 / 5 = 2` tokens (integer division rounds down from 2.4)
5. Total actually transferred: 3 + 2 = 5 tokens
6. For bribe approval or final treasury transfer, the code uses `balance - treasuryFee = 30 - 6 = 24`
7. However, the actual tokens remaining should be `30 - 5 = 25` tokens

This means the contract approves or transfers 24 tokens when there are actually 25 tokens available.

Recommendation

Calculate the sum to transfer to the `treasury` and `developer`, then subtract that value from the `balance` for the following `approve()` and `transfer()`.

Team Response

Fixed.

[I-01] Excessive Gas Consumption in Multiple Tool Purchases

Severity

Informational

Description

The `purchaseTool()` function in the Factory contract contains a gas inefficiency issue. When a user purchases multiple tools at once using the `toolAmount` parameter, the function burns tokens individually for each tool within a loop instead of calculating the total cost and burning tokens once at the end.

```
function purchaseTool(uint256 tokenId, uint256 toolId, uint256 toolAmount
    ) external nonReentrant tokenExists(tokenId) {
     // ... existing code ...
     for (uint256 i = 0; i < toolAmount; i++) {
         // ... existing code ...
         IUnits(units).burn(msg.sender, cost);
         // @audit-issue - Burns tokens in each loop iteration
     }
}
```

[language=solidity]

This approach results in unnecessary gas consumption as each `burn` operation triggers an external call, which is significantly more expensive than making a single external call after accumulating the total amount to burn.

Location of Affected Code

File: Factory.sol#L85-L98

Impact

Unnecessary gas consumption

Recommendation

The function should accumulate the total cost through the loop and perform a single `burn` operation after the loop completes, which would substantially reduce gas costs for users purchasing multiple tools.

Team Response

Fixed.

shieldify

Thank you!