TrickOrTreat Auction Audit Report

Version 1.0

0xBlockPay

November 14, 2024

TrickOrTreat Auction Audit Report

0xBlockPay

October 24, 2024

Prepared by: 0xBlockPay

Lead Security Researcher: - eth0x

Table of Contents

Protocol Summary

SpookySwap is a Halloween-themed decentralized application where users can participate in a thrilling "Trick or Treat" experience! Swap ETH for special Halloween-themed NFT treats. But beware, you might get tricked! There's a small chance your treat will cost half the price, or you might have to pay double. Collect rare NFTs, trade them with friends, or hold onto them for spooky surprises. Will you be tricked or treated?

Disclaimer

The 0xBlockPay team makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by the team is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the Solidity implementation of the contracts.

Risk Classification

		Impact		
		High	Medium	Low
	High	H	H/M	M
Likelihood	Medium	H/M	M	M/L
	Low	M	M/L	L

We use the CodeHawks severity matrix to determine severity. See the documentation for more details.

Audit Details

Scope

The findings describe in this document correspond the following repository and commit hash:

```
git clone https://github.com/Cyfrin/2024-10-trick-or-treat.git
```

```
Commit Hash:
```

In Scope:

```
src/
#-- SpookySwap.sol
```

- Blockchains: EVM Equivalent Chains Only
- Tokens: Native ETH

Roles

- Owner/Admin (Trusted) - Can add new treats, set treat costs, and withdraw collected fees.
- User/Participant - Can swap ETH for Halloween treat NFTs, experience “Trick or Treat”, and trade NFTs with others.

Executive Summary

*We spend 10 hours with 1 auditor using foundry, aderyn and slither tools.

Issues found

Severity	Number of issues found
High	0
Medium	1
Low	4
Info	0
Gas	0
Total	5

Findings

Medium

[M-1] Use of `_mint` Instead of `_safeMint` for ERC721 Tokens

Description:

The contract uses the `_mint` function to create new ERC721 tokens, which does not check if the recipient is capable of receiving ERC721 tokens.

Instances:

1. **Minting to Contract Itself (Line 81):**

```
1 _mint(address(this), tokenId);
```

2. **Minting to Recipient (Line 110):**

```
1 _mint(recipient, tokenId);
```

Impact:

- **Potential Token Loss:**
  - If tokens are minted to a contract that does not implement the `onERC721Received` function, they could become permanently locked, leading to loss of tokens.
- **Compliance with ERC721 Standard:**
  - The ERC721 standard recommends using `_safeMint` to ensure safe transfers to contracts.

Recommendation:

- **Use `_safeMint` Function:**
  - Replace `_mint` with `_safeMint` to include safety checks.
  - **Updated Minting to Contract Itself:**

```
1 _safeMint(address(this), tokenId);
```

  - **Updated Minting to Recipient:**

```
_safeMint(recipient, tokenId);
```

- **Handle Potential Reverts:**
  - Be prepared for the possibility that `_safeMint` may revert if the recipient cannot handle ERC721 tokens.
- **Inform Users:**
  - Clearly communicate to users that their addresses must be capable of receiving ERC721 tokens, especially if they are using smart contract wallets.

Low

[L-1] Gas-Limited ETH Transfer in withdrawFees

Summary

- Root Cause: Usage of `transfer()` which has a hard gas limit of 2300 gas
- Impact: Funds could be permanently locked if the owner is a smart contract (e.g., Gnosis Safe, other smart wallets)

Vulnerability Details

https://github.com/Cyfrin/2024-10-trick-or-treat/blob/main/src/TrickOrTreat.sol#L146-L150

```
function withdrawFees() public onlyOwner {
        uint256 balance = address(this).balance;
      /// @audit: use of transfer is probelamatic for smart wallets
        payable(owner()).transfer(balance);
        emit FeeWithdrawn(owner(), balance);
    }
```

The `withdrawFees()` function uses the `transfer()` method to send ETH to the owner, However it is problematic because:

1. `transfer()` has a hard gas limit of 2300 gas
2. Smart contract wallets (like Gnosis Safe) typically require more than 2300 gas to process incoming ETH
3. Modern smart wallets commonly used as multi-sigs would fail to receive funds
4. The operation would revert, effectively locking the fees in the contract

POC

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.24;
import "forge-std/Test.sol";
import "../src/SpookySwap.sol";

contract GnosisSafeMock is Test {
    // Simulate Gnosis Safe receive function that uses more than 2300 gas
    receive() external payable {
        // Perform some operations that consume gas
        uint256 a = 1;
        for(uint i = 0; i < 10; i++) {
            a = a * 2;
        }
    }
}

contract SpookySwapTest is Test {
    SpookySwap spookySwap;
    GnosisSafeMock safeWallet;

    function setUp() public {
        SpookySwap.Treat[] memory treats = new SpookySwap.Treat[]();
        treats[0] = SpookySwap.Treat("Candy", 1 ether, "uri");
        spookySwap = new SpookySwap(treats);
        safeWallet = new GnosisSafeMock();
    }

    function testWithdrawToSmartWallet() public {
        // First, let's add some ETH to the contract
        address(spookySwap).call{value: 1 ether}("");

        // Change owner to our mock Gnosis Safe
        spookySwap.changeOwner(address(safeWallet));

        // This will fail because transfer() can't handle smart contract wallet
        vm.expectRevert();
        spookySwap.withdrawFees();
    }
}
```

Impact

Funds will be locked in contract if owner is a smart wallet or contract that requires more than 2300 gas to receive ether

Tools Used

Manual Review

Recommendations

Replace `transfer()` with the recommended `call()` pattern:

```
function withdrawFees() public onlyOwner {
    uint256 balance = address(this).balance;

    (bool success, ) = payable(owner()).call{value: balance}("");
    require(success, "Fee withdrawal failed");

    emit FeeWithdrawn(owner(), balance);
  }
```

[L-2] Incorrect Event Emission for Insufficient Payment in NFT Minting

01. Relevant GitHub Links

- https://github.com/Cyfrin/2024-10-trick-or-treat/blob/9cb3955058cad9dd28a24eb5162a96d759bfa842/src/TrickOrTreat.sol#L77C1-L89C63

02. Summary

The contract emits a Swapped event even when a user has not sent enough ETH to complete the transaction. In such cases, the NFT is minted to the contract and marked as pending, rather than being directly swapped. This incorrect event can cause confusion by suggesting the transaction succeeded and the NFT was swapped to the user, while it's actually still pending.

03. Vulnerability Details

When a user sends insufficient ETH, the contract mints an NFT to itself and marks it as pending by recording the user's address and the amount paid. However, the contract still emits the Swapped event, which falsely signals that the transaction was successful and that the NFT was swapped to the user. This misrepresentation can lead to user confusion or misunderstandings about the transaction status.

```
} else {
    // User didn't send enough ETH
    // Mint NFT to contract and store pending purchase
    uint256 tokenId = nextTokenId;
    _mint(address(this), tokenId);
    _setTokenURI(tokenId, treat.metadataURI);
    nextTokenId += 1;

    pendingNFTs[tokenId] = msg.sender;
    pendingNFTsAmountPaid[tokenId] = msg.value;
    tokenIdToTreatName[tokenId] = _treatName;

    emit Swapped(msg.sender, _treatName, tokenId);
```

This event should only be emitted if the transaction completes successfully and the NFT is actually transferred to the user, not when it's stored as a pending NFT.

03. Impact

- User Confusion and Misinterpretation: The Swapped event indicates a successful transaction, which can mislead users into thinking their purchase was completed when, in reality, the NFT is still pending.
- Potential Application Logic Errors: External systems or interfaces relying on events for status updates may misinterpret the transaction status, leading to potential errors in application logic or user interfaces.

04. Proof of Concept

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.24;

import {Test, console} from "forge-std/Test.sol";
import {SpookySwap} from "../src/TrickOrTreat.sol";

contract TrickOrTreatTest is Test {
    SpookySwap spookyswapInstance;

    address owner;
    address user;

    function setUp() public {
        owner = vm.addr(0x1);
        user = vm.addr(0x2);

```

```
17          vm.label(owner, "Owner");
18          vm.deal(owner, 100 ether);
19
20          vm.label(user, "User");
21          vm.deal(user, 100 ether);
22
23          SpookySwap.Treat[] memory TreatList;
24
25          vm.prank(owner);
26          spookyswapInstance = new SpookySwap(TreatList);
27      }
28
29      function test_IncorrectEvent() public {
30          vm.prank(owner);
31          spookyswapInstance.addTreat("Example", 1 ether, "test");
32
33          uint256 tokenId = spookyswapInstance.nextTokenId();
34
35          // random = 2
36          // double price
37          vm.warp(123456790);
38          vm.prevrandao(987654324);
39
40          vm.expectEmit(true, true, true, true);
41                  // Emits an event indicating the treat was swapped to
                       the user
42          emit SpookySwap.Swapped(user, "Example", tokenId);
43          vm.prank(user);
44          // Insufficient funds: treat purchase is pending until full
               payment is made
45          spookyswapInstance.trickOrTreat{value: 1 ether}("Example");
46
47          // user have no nft
48          vm.assertEq(0, spookyswapInstance.balanceOf(user));
49      }
50  }
```

05. Tools Used

Manual Code Review and Foundry

06. Recommended Mitigation

1. Emit a Different Event for Pending Transactions: Introduce a new event, such as PendingSwap, specifically for cases where the NFT is stored in the contract due to insufficient payment. This provides clarity on the transaction's actual status.

```
emit PendingSwap(msg.sender, _treatName, tokenId);
```

[L-03] Potential Integer Division Issues Leading to Incorrect Cost Calculations

The contract performs integer division in its cost calculations, which can lead to rounding errors and incorrect pricing. Since Solidity uses integer arithmetic, any fractional components resulting from division are truncated. This can cause users to be overcharged or undercharged when purchasing treats, leading to financial discrepancies and user dissatisfaction.

Vulnerability Details

In the `trickOrTreat` function, the cost of a treat is adjusted based on a random multiplier:

```
uint256 requiredCost = (treat.cost * costMultiplierNumerator) /
    costMultiplierDenominator;
```

When `costMultiplierNumerator` and `costMultiplierDenominator` result in a fraction (e.g., 1/2 for half price), integer division in Solidity truncates any decimal values. This means that the calculated `requiredCost` may be less than or greater than the intended value due to rounding down.

Example Scenario:

- If a treat costs 5 wei and the user gets a half-price discount, the calculation becomes `(5 * 1) / 2 = 2` wei.
- The actual half price should be 2.5 wei, but due to integer division, the user is undercharged by 0.5 wei.
- Conversely, if the treat cost is 7 wei, half price would ideally be 3.5 wei, but the calculation yields 3 wei, undercharging the user by 0.5 wei.

Impact

- **Financial Discrepancies:** Users may be overcharged or undercharged, leading to loss of funds for either the user or the contract owner.
- **User Dissatisfaction:** Inconsistent pricing can erode trust in the platform, resulting in a negative user experience.
- **Accounting Issues:** Over time, these small discrepancies can accumulate, causing significant financial misalignments in the contract's bookkeeping.

Proof of Concept

1. **Undercharging Scenario:**
    - Treat cost: 5 wei
    - Multiplier: Half price (1/2)
    - Calculation: `(5 * 1) / 2 = 2` wei
    - Intended cost: 2.5 wei
    - Actual cost charged: 2 wei
    - Undercharge: 0.5 wei
2. **Overcharging Scenario:**
    - Treat cost: 5 wei
    - Multiplier: Double price (2/1)
    - Calculation: `(5 * 2) / 1 = 10` wei
    - Intended cost: 10 wei (correct in this case)

    Since the double price scenario doesn't involve fractions, it calculates correctly. The issue mainly arises when the result should be a fractional value.

Recommendations

- **Use Fixed-Point Arithmetic:**

  Implement fixed-point math to handle decimal values accurately. You can multiply all cost values by a scaling factor (e.g., 1e18) to preserve precision during calculations.

```
uint256 scalingFactor = 1e18;
uint256 requiredCost = (treat.cost * scalingFactor *
    costMultiplierNumerator) / (costMultiplierDenominator *
    scalingFactor);
```

- **Explicit Rounding Logic:**

  Decide on a rounding strategy (e.g., round up to the nearest wei) and implement it in the calculation to ensure predictable and fair pricing.

```
uint256 requiredCost = (treat.cost * costMultiplierNumerator +
    costMultiplierDenominator - 1) / costMultiplierDenominator;
```

- **Validate Cost Calculations:**

  After computing `requiredCost`, include a check to ensure it aligns with expected values, preventing significant deviations.

- **Inform Users:**

  Clearly document any rounding behavior so users are aware of how prices are calculated.

[L-4] Unsafe NFT transfer

Summary

The SpookySwap contract uses unsafe NFT transfer methods (`_mint` and `_transfer`) instead of their safe counterparts (`_safeMint` and `_safeTransfer`). This can lead to permanent loss of NFTs when they are transferred to contracts that don't support ERC721 token reception.

Vulnerability Details

The contract implements NFT transfers in two critical functions:

1. In the `mintTreat` internal function:

```
function mintTreat(address recipient, Treat memory treat) internal {
    uint256 tokenId = nextTokenId;
    _mint(recipient, tokenId);  // Unsafe minting
    _setTokenURI(tokenId, treat.metadataURI);
    nextTokenId += 1;
}
```

1. In the `resolveTrick` function:

```
function resolveTrick(uint256 tokenId) public payable nonReentrant {
    // ... payment validation ...
    _transfer(address(this), msg.sender, tokenId);  // Unsafe transfer
    // ... cleanup code ...
}
```

The vulnerability arises because these functions don't verify whether the recipient can handle ERC721 tokens. According to the ERC721 standard, contracts must implement the `onERC721Received` function to receive tokens. Without this check, tokens can be permanently locked in contracts that don't support ERC721.

Example of a vulnerable scenario:

```
contract VulnerableContract {
    function buyTreat(address spookySwap, string memory treatName)
       external payable {
        SpookySwap(spookySwap).trickOrTreat(treatName);
        // NFT will be lost as this contract cannot handle ERC721
           tokens
```

```
    }
}
```

Impact

- Permanent loss of NFTs when transferred to incompatible contracts
- No recovery mechanism for lost tokens
- Potential financial loss for users who accidentally send NFTs to contract addresses
- Deterioration of user trust in the platform

Tools Used

- Manual review

Recommendations

1. Replace unsafe transfer methods with their safe counterparts:

```
// In mintTreat function
function mintTreat(address recipient, Treat memory treat) internal {
    uint256 tokenId = nextTokenId;
    _safeMint(recipient, tokenId);  // Safe minting
    _setTokenURI(tokenId, treat.metadataURI);
    nextTokenId += 1;
}

// In resolveTrick function
function resolveTrick(uint256 tokenId) public payable nonReentrant {
    // ... payment validation ...
    _safeTransfer(address(this), msg.sender, tokenId);  // Safe
        transfer
    // ... cleanup code ...
}
```