Scanned Code Report

AUDITAGENT

Powered by NETHERMIND

Code Info

Organization ny423	Scan ID 1
Repository animoca-pixels	Branch main
Commit Hash fc623bc8837bec49e4d09c285be41e1573eb96b9	

Contracts in scope

`contracts/src/DepositThresholdContract.sol`

Code Statistics

Findings	Contracts Scanned	Lines of Code
11	1	135

Code Summary

The DepositThresholdContract is a smart contract designed to manage a two-week whitelist campaign where users can deposit ERC20 tokens to gain whitelist access. The contract implements a dynamic pricing mechanism that increases the required deposit amount over time, creating urgency and incentivizing early participation.

Key Features

- **Time-Limited Campaign**: The campaign runs for exactly 14 days from a predetermined starting timestamp.
- **Daily Whitelist Limits**: Each day has a fixed maximum number of whitelist spots available.
- **Progressive Pricing Structure**: The required deposit amount increases each day:
- Week 1: Starts at the base amount and increases by 10% of the base amount each day.
- Week 2: Starts at double the base amount and increases by 20% of the base amount each day.
- **Admin Withdrawal**: Contract owner can withdraw deposited tokens to any address.
- **Security Measures**: Implements reentrancy protection and ownership controls.

Technical Implementation

- The contract inherits from OpenZeppelin's `Ownable` and `ReentrancyGuard` contracts.
- Uses SafeERC20 for secure token transfers.
- Stores user deposits and tracks daily whitelist counts.
- Provides utility functions to check campaign status and calculate required deposit amounts.

Entry Points and Actors

- **depositToken()**: Called by **users** who want to participate in the whitelist. This function:
- Verifies the campaign is active and daily limits aren't reached
- Calculates the required deposit amount based on the current day
- Transfers tokens from the user to the contract
- Updates the user's deposit record and the daily whitelist count

- Emits a DepositedToken event

- **adminWithdraw(uint256 _amount, address _to)**: Called by the **contract owner** to withdraw tokens from the contract to a specified address.

1 of 11 Findings DepositThresholdContract.sol

Incorrect implementation of isCampaignActive function ● High Risk

The `isCampaignActive()` function is implemented incorrectly. Instead of returning a boolean value based on the timestamp checks, it uses `require` statements which will revert the transaction if the conditions are not met. This means that any function that calls `isCampaignActive()` expecting a boolean return value will never receive `false` - it will either receive `true` or the transaction will revert.

```
function isCampaignActive() public view returns (bool) {
    require(block.timestamp >= startingTimestamp, "Campaign is not active");
    require(block.timestamp < startingTimestamp + CAMPAIGN_DURATION,
"Campaign has ended");
    return true;
}
```

This implementation breaks the expected behavior of the function and violates the "Campaign Time Window Invariant" which states that the function should return a boolean indicating whether the campaign is active.

The impact of this issue is severe because:

1. The `depositToken()` function calls `isCampaignActive()` and expects it to return a boolean, but it will revert instead if the campaign is not active.
2. Any external contracts or interfaces that might rely on calling this view function to check the campaign status will not work as expected.
3. It makes the function name misleading - a function named `isCampaignActive()` should return a boolean, not revert.

This could lead to unexpected transaction failures and broken integrations with other contracts or front-end applications that expect to be able to check the campaign status without reverting.

Additional Insights: The function design fundamentally conflicts with its intended purpose. When a function returns a boolean type, callers expect to receive either true or false, not a transaction revert. The current implementation creates a misleading API that will cause integration failures with any external systems. This pattern breaks the principle of least surprise and violates function signature expectations. The severity remains high because this isn't just a code style issue - it creates a situation where the function cannot be used as its signature suggests, potentially breaking any system that integrates with this contract. A proper implementation would perform the timestamp checks and return the appropriate boolean value without reverting.

2 of 11 Findings DepositThresholdContract.sol

Potential for admin to withdraw all tokens, including user deposits ● Medium Risk

The `adminWithdraw` function allows the contract owner to withdraw any amount of tokens from the contract, without any restrictions. This means that the owner could potentially withdraw all tokens, including those deposited by users, even before the campaign ends.

```
function adminWithdraw(uint256 _amount, address _to) external onlyOwner
nonReentrant {
    require(_to != address(0), "Invalid recipient address");
    require(token.transfer(_to, _amount), "Token transfer failed");
    emit AdminWithdrawal(_amount, _to);
}
```

This issue is problematic because:

1. There is no mechanism to ensure that user deposits are protected until the campaign ends.
2. Users have no guarantee that their deposits will be used as intended for the whitelist campaign.
3. The contract owner has unrestricted access to all tokens in the contract, which creates a centralization risk.

While this may be an intended design choice, it represents a significant trust assumption that users must place in the contract owner. If the owner's private key is compromised, or if the owner acts maliciously, all user deposits could be at risk. This vulnerability affects the security model of the contract and could undermine user trust in the whitelist campaign.

Additional Insights: The unrestricted admin withdrawal capability creates a significant trust assumption that users must accept. While the contract implements proper access controls through the Ownable pattern and includes the nonReentrant modifier for security, there are no time-locks, withdrawal limits, or other safeguards to protect user deposits. The contract's transparency about this functionality (public adminWithdraw function and emitted events) mitigates some concerns, but the centralization risk remains substantial. This design may be intentional for the campaign, but represents a significant trust requirement that users should be aware of.

3 of 11 Findings DepositThresholdContract.sol

Risky Strict Equality Check

● Medium Risk

The `getRemainingWhitelistsToday()` function in DepositThresholdContract.sol (line 128-134) uses a dangerous strict equality check:

```
currentDay == 0 || currentDay > 14
```

Strict equality checks (==) on values that determine control flow can be risky, especially when dealing with time-based calculations. In this case, the contract is checking if `currentDay` is exactly 0 or greater than 14. This could potentially lead to unexpected behavior if there are any calculation errors or edge cases in determining the current day.

If the calculation of `currentDay` is slightly off due to block timestamp variations or other factors, the condition might not evaluate as expected, potentially causing the contract to behave incorrectly.

Lack of user deposit limit check

● Low Risk

The `depositToken()` function does not check if a user has already made a deposit. This means that a single user can make multiple deposits and take up multiple whitelist spots, potentially preventing other users from participating in the campaign.

```
function depositToken() external nonReentrant {
    uint256 currentDay = getCurrentDay();
    require(isCampaignActive(), "Campaign is not active");
    require(dailyWhitelistCount[currentDay] < whitelistsPerDay, "Daily
whitelist limit reached");

    uint256 requiredAmount = getRequiredDepositAmount(currentDay);
    require(token.transferFrom(msg.sender, address(this), requiredAmount),
"Transfer failed");
    userTokenDeposits[msg.sender] += requiredAmount;
    dailyWhitelistCount[currentDay] += 1;

    emit DepositedToken(msg.sender, requiredAmount, currentDay);
}
```

The function increments `userTokenDeposits[msg.sender]` without checking if the user has already made a deposit. It also increments `dailyWhitelistCount[currentDay]` for each deposit, which means a single user can consume multiple whitelist spots.

This could lead to a situation where a small number of users monopolize the whitelist spots, defeating the purpose of having a fair distribution mechanism. It could also be exploited by malicious users who want to prevent others from participating in the campaign.

Additional Insights: The contract's design appears to intentionally allow users to make multiple deposits, creating a market-driven mechanism where users can secure multiple whitelist spots by paying more. The daily limit through whitelistsPerDay and dailyWhitelistCount prevents complete monopolization of spots. The absence of a simple user deposit limit check (which would be trivial to implement) suggests this is a deliberate economic model rather than an oversight. This aligns with the increasing price model implemented in getRequiredDepositAmount(). Without explicit requirements stating users should be limited to one deposit, this is likely a design choice rather than a vulnerability.

Incorrect calculation in getRequiredDepositAmount function

● Low Risk

The `getRequiredDepositAmount()` function contains a calculation error that could lead to incorrect deposit amounts being required from users.

```
function getRequiredDepositAmount(uint256 _day) public view returns (uint256)
{
    uint256 weekConst = 8;
    require(_day >= 1 && _day <= 14, "Invalid day");
    uint256 baseAmount = roninTokenAmount;
    uint256 week = (_day <= weekConst) ? 1 : 2;
    uint256 dayInWeek = (_day <= weekConst) ? _day : _day - weekConst;
    if (week == 1) {
        return baseAmount * 1 + baseAmount * (dayInWeek - 1) * 10 / 100;
    } else {
        return baseAmount * 2 + baseAmount * (dayInWeek - 1) * 20 / 100;
    }
}
```

The issue is in the calculation for week 1. The expression `baseAmount * 1 + baseAmount * (dayInWeek - 1) * 10 / 100` can be simplified to `baseAmount + baseAmount * (dayInWeek - 1) * 10 / 100`. However, due to Solidity's integer division, when `(dayInWeek - 1) * 10` is less than 100, the division by 100 will result in 0, effectively making the second term disappear.

For example, on day 1 of week 1, `dayInWeek` is 1, so `(dayInWeek - 1) * 10` is 0, and `0 / 100` is 0. This means the required deposit amount will be just `baseAmount`, which is correct.

But on day 2 of week 1, `dayInWeek` is 2, so `(dayInWeek - 1) * 10` is 10, and `10 / 100` is 0 due to integer division. This means the required deposit amount will still be just `baseAmount`, which is incorrect - it should be `baseAmount * 1.1`.

This issue affects days 2-9 of the campaign, where the required deposit amount will be incorrectly calculated, leading to users depositing less than they should according to the intended pricing structure.

Additional Insights: The integer division issue in getRequiredDepositAmount() is valid but context-dependent. For tokens with small decimal places or when roninTokenAmount is set to a small value (less than 10), the percentage calculations would round down to zero due to Solidity's integer division, causing incorrect pricing tiers. While this would work correctly with sufficiently large token amounts (as is common with most ERC20 tokens), the function lacks safeguards to ensure roninTokenAmount is large enough for the percentage calculations to be meaningful. This creates a potential for misconfiguration that could break the intended pricing structure.

Lack of mechanism to handle campaign end

● Low Risk

The contract does not have a mechanism to handle what happens after the campaign ends. There is no functionality to distribute whitelist spots, return unused deposits, or transition to a post-campaign state.

The campaign duration is defined as 14 days:

```
uint256 constant CAMPAIGN_DURATION = 14 days; // Two-week campaign duration
```

And the `isCampaignActive` function checks if the current time is within this duration:

```
function isCampaignActive() public view returns (bool) {
    require(block.timestamp >= startingTimestamp, "Campaign is not active");
    require(block.timestamp < startingTimestamp + CAMPAIGN_DURATION,
"Campaign has ended");
    return true;
}
```

However, there is no function or mechanism that is triggered when the campaign ends. This means that:

1. User deposits remain locked in the contract indefinitely after the campaign ends.
2. There is no clear way for users to claim their whitelist spots or benefits after the campaign.
3. The contract owner can still withdraw all tokens after the campaign ends, but there's no structured process for this.

This lack of a post-campaign mechanism could lead to confusion and uncertainty for users, as they have no clear understanding of what happens to their deposits or whitelist status after the campaign ends. It also places additional trust requirements on the contract owner to handle the post-campaign process fairly and transparently.

Additional Insights: The minimalist approach to campaign end handling follows a common pattern in whitelist campaigns. The contract effectively freezes participation after the campaign period through the isCampaignActive check, creating a permanent record of participants in the userTokenDeposits mapping. This separation of concerns is a reasonable design choice, where on-chain components handle deposit collection and participant recording, while post-campaign processes (token distribution, etc.) would be managed separately. However, this design creates some uncertainty for users about what happens to their deposits after the campaign ends.

7 of 11 Findings

DepositThresholdContract.sol

Non-Specific Solidity Pragma Version

● Low Risk

The contract uses a floating pragma statement:

```
pragma solidity ^0.8.20;
```

Using a floating pragma (^) means the contract can be compiled with any compatible compiler version, which might introduce inconsistencies in behavior or unexpected bugs if compiled with a different version than intended.

It's recommended to lock the pragma to a specific version (e.g., `pragma solidity 0.8.20;`) to ensure the contract is always compiled with the same compiler version, providing consistent behavior across all environments.

Magic Numbers Instead Of Constants

● Low Risk

The contract uses magic numbers in multiple places instead of defining them as named constants:

```
if (currentDay == 0 || currentDay > 14) {
```

The number `14` appears multiple times in the code and represents what seems to be the maximum number of days for the whitelist period. Using magic numbers makes the code less readable and more error-prone if these values need to be changed in the future.

These values should be defined as named constants at the contract level, such as:

```
uint256 private constant MAX_WHITELIST_DAYS = 14;
```

This would improve code readability, maintainability, and reduce the risk of inconsistencies if the value needs to be updated.

9 of 11 Findings DepositThresholdContract.sol

Incorrect `nonReentrant` modifier Placement

● Low Risk

In the `adminWithdraw` function, the `nonReentrant` modifier is placed after the `onlyOwner` modifier:

```
function adminWithdraw(uint256 _amount, address _to) external onlyOwner
nonReentrant {
```

Best practice is to place the `nonReentrant` modifier first, before any other modifiers. This ensures that the reentrancy guard is checked before executing any other modifier logic, which could potentially contain external calls or state changes that might be vulnerable to reentrancy attacks.

The correct order should be:

```
function adminWithdraw(uint256 _amount, address _to) external nonReentrant
onlyOwner {
```

10 of 11 Findings DepositThresholdContract.sol

PUSH0 Opcode Compatibility Issue

● Low Risk

The contract uses Solidity version 0.8.20 which targets the Shanghai EVM version by default:

```
pragma solidity ^0.8.20;
```

This compiler version generates bytecode that includes the PUSH0 opcode, which was introduced in the Shanghai upgrade. This opcode is not supported on all chains, particularly some Layer 2 solutions or other EVM-compatible blockchains that haven't implemented the Shanghai upgrade.

If the contract is intended to be deployed on chains other than Ethereum mainnet, it might fail to deploy due to the unsupported opcode. To ensure compatibility with all chains, either:

1. Use an earlier Solidity version (< 0.8.20)
2. Explicitly specify the EVM version in the compiler settings (e.g., `evmVersion: "paris"` in the compiler configuration)

11 of 11 Findings DepositThresholdContract.sol

Non-Immutable State Variables

● Best Practices

Several state variables in the DepositThresholdContract are not marked as immutable despite being set only once in the constructor and never modified afterward:

1. `token` (line 14) - The ERC20 token used for deposits
2. `startingTimestamp` (line 15) - The timestamp when the contract starts operating
3. `roninTokenAmount` (line 16) - The amount of tokens required for deposit
4. `whitelistsPerDay` (line 17) - The number of whitelists allowed per day

Marking these variables as `immutable` would optimize gas usage as immutable variables are stored in the contract bytecode rather than storage. This can save approximately 20,000 gas during deployment and reduce gas costs for functions that read these variables.

Disclaimer