Scanned Code Report

AUDITAGENT

Code Info

Free Scan

Organization
ISP-Team-Internal

Scan ID
2

Repository
aiveronica-contracts

Branch
main

Commit Hash
c2e4952943cc530a5e3319445f47030613b283bc

Contracts in scope

src/DepositThresholdNFT.sol
script/DeployDepositThresholdNFT.s.sol

Code Statistics

Findings
16

Contracts Scanned
2

Lines of Code
319

Code Summary

The DepositThresholdNFT protocol implements a time-limited whitelist campaign where users can mint non-transferable ERC721 tokens by depositing specific amounts of an ERC20 token. The protocol operates over a predefined campaign duration divided into daily periods, with each day having its own deposit threshold and whitelist limit.

Key features of the protocol include:

- Time-based campaign structure with a defined start time and duration
- Daily deposit thresholds that typically increase over time
- Daily whitelist limits to control the number of participants per day
- Non-transferable NFTs that are minted upon successful deposit
- Prevention of multiple mints by the same user on the same day
- Admin withdrawal functionality for collected ERC20 tokens

The protocol is designed with several safeguards:
- Reentrancy protection for sensitive functions
- Ownership controls for administrative actions
- Comprehensive input validation during initialization
- Checks for campaign activity status before allowing mints

Main Entry Points and Actors

Entry Points:

1. `mint()`: Allows users to deposit the required ERC20 tokens for the current day and receive an NFT. The function checks if the daily whitelist limit hasn't been reached, if the user hasn't already minted today, and if the user has sufficient token balance.

Actors:

1. **Users**: Regular participants who can mint NFTs by depositing the required amount of ERC20 tokens.
2. **Admin/Owner**: The contract owner who can withdraw collected ERC20 tokens from the contract.

The protocol implements a dynamic pricing model where the required deposit amount typically increases each day of the campaign, creating a sense of urgency and potentially rewarding early participants with lower entry costs. The daily whitelist limits also provide a mechanism to control the rate of participation and create scarcity.

No emergency pause mechanism

Medium Risk

The contract does not include an emergency pause mechanism that would allow the owner to temporarily halt minting in case of discovered vulnerabilities or other issues. Once the campaign starts, it will continue until its scheduled end with no way to stop it early.

```
modifier campaignActive() {
    require(block.timestamp >= startingTimestamp, "Campaign is not active");
    require(
        block.timestamp < startingTimestamp + campaignDuration,
        "Campaign has ended"
    );
    _;
}
```

If a critical issue is discovered during the campaign, there's no way for the owner to pause minting operations to protect users or the protocol. The lack of a pause mechanism reduces the owner's ability to respond to unforeseen circumstances, potentially leaving the contract vulnerable for the duration of the issue.

Additional Insights: The absence of an emergency pause mechanism represents a significant operational limitation. If vulnerabilities are discovered after deployment, the contract owner has no way to temporarily halt operations until the campaign naturally ends. This design choice creates a rigid system where any discovered issues must remain exploitable for the duration of the campaign. While this doesn't introduce a vulnerability directly, it removes a critical safety mechanism commonly implemented in DeFi protocols. The impact is somewhat mitigated by the time-limited nature of campaigns, but for longer campaigns, this limitation could expose users to extended periods of risk if issues are discovered.

Minting might fail due to token decimals mismatch

Medium Risk

The contract implicitly assumes the ERC20 token has 18 decimals, as evident in the deployment script, but doesn't verify or store this information. If a token with fewer decimals is used, this can lead to unexpectedly high deposit requirements.

```
// From the deployment script
dailyTokenAmounts[0] = 60 * 10 ** 18; // Day 1: 60 tokens
```

For example, if a token with 6 decimals (like USDC) is used, setting an amount of `60 * 10**18` would actually require 60 * 10^12 tokens, which is significantly more than intended. This mismatch could make minting NFTs economically impossible if the deposit thresholds are unexpectedly high due to incorrect decimal handling.

Additional Insights: The contract's implicit assumption of 18 decimal places for all ERC20 tokens creates a significant risk when using tokens with different decimal configurations. For tokens with fewer decimals (like USDC with 6), the deposit requirements would be astronomically higher than intended - potentially by a factor of 10^12. This would make participation economically impossible for most users. The issue is particularly concerning because there's no validation or adjustment mechanism in the contract to handle different token decimals. This represents a fundamental design flaw that could render the entire contract unusable if deployed with the wrong token, requiring a complete redeployment rather than a simple parameter adjustment.

No validation that daily token amounts are not zero

Low Risk

The contract constructor validates that all daily token amounts are greater than zero. However, there's no validation that the token amounts aren't trivially small. For tokens with 18 decimal places, an amount of 1 (1e-18 tokens) would pass the validation but would effectively be a zero amount practically.

```
for (uint256 i = 0; i < _dailyTokenAmounts.length; i++) {
    require(
        _dailyTokenAmounts[i] > 0,
        "Daily token amount must be greater than 0"
    );
    // ...
}
```

This could potentially allow users to mint NFTs essentially for free on certain days if the campaign creator mistakenly sets very small amounts. While the impact is limited to lost revenue rather than security vulnerabilities, it could undermine the economic model of the campaign.

Additional Insights: The validation issue extends beyond just zero values to include trivially small amounts. For tokens with 18 decimals, an amount of 1 wei (1e-18 tokens) would technically pass the validation but represent an economically insignificant amount. This creates a potential economic vulnerability where users could mint NFTs for virtually no cost if the campaign creator mistakenly sets very small thresholds. While this wouldn't result in direct fund loss, it could completely undermine the token deposit mechanism that forms the core economic model of the campaign. The risk is mitigated by the fact that campaign parameters are set by the owner at deployment and cannot be changed afterward, making this primarily a deployment configuration concern rather than an ongoing security risk.

Potential gas limit concerns for long campaigns

Low Risk

The constructor initializes arrays for daily token amounts and whitelist limits by pushing values in a loop:

```
for (uint256 i = 0; i < _dailyTokenAmounts.length; i++) {
    dailyTokenAmounts.push(_dailyTokenAmounts[i]);
    dailyWhitelistLimits.push(_dailyWhitelistLimits[i]);
}
```

If the campaign duration is very long (e.g., hundreds of days), this loop could potentially exceed the block gas limit during contract deployment. This would prevent the contract from being deployed for long-duration campaigns. While the intended two-week campaign (14 days) won't encounter this issue, it's a limitation worth noting for any potential longer campaigns.

Additional Insights: The array initialization approach in the constructor creates a hard upper limit on campaign duration due to gas constraints. While the intended two-week campaign duration is well within safe limits, any attempt to deploy with significantly longer durations (hundreds of days) would likely exceed block gas limits. This represents a scalability limitation rather than a security vulnerability. The contract would simply fail to deploy with excessive parameters rather than creating an exploitable condition. Since the contract is designed for time-limited campaigns and the intended duration is well within safe parameters, this limitation primarily affects potential future use cases rather than the current implementation.

Using `_safeMint` prevents contract wallets from participating

• Low Risk

The `mint()` function uses OpenZeppelin's `_safeMint`, which calls `onERC721Received` on the recipient if it is a contract. Contracts (including Gnosis Safe, multisigs, or other smart contracts) that do not implement `IERC721Receiver` will cause `_safeMint` to revert, blocking them from participating in the campaign.

```
// in mint():
token.safeTransferFrom(msg.sender, address(this), requiredAmount);
_safeMint(msg.sender, tokenId);
```

This design unintentionally limits participation to EOA addresses only. If a user calls `mint()` from a contract that does not implement the ERC-721 receiver interface, the mint will revert and the deposit will not succeed.

Additional Insights: The use of _safeMint creates an unintentional restriction that prevents contract wallets from participating in the campaign. This affects increasingly popular wallet types including Gnosis Safe multisigs, smart contract wallets, and other contract-based accounts that don't implement IERC721Receiver. The restriction is particularly problematic as these wallet types are commonly used by larger investors and DAOs who might be interested in participating. While this doesn't create a security vulnerability, it significantly limits the potential user base and creates an artificial barrier to participation. The impact is primarily on adoption and inclusivity rather than security of the protocol itself.

No mechanism to retrieve accidentally sent tokens

Low Risk

The contract only has a withdrawal mechanism for the configured ERC20 token. If other tokens (including ETH/native currency) are accidentally sent to the contract address, there's no way to recover them.

```
function adminWithdraw(
    uint256 _amount,
    address _to
) external nonReentrant onlyOwner {
    require(_to != address(0), "Invalid recipient address");
    require(_amount > 0, "Amount must be greater than 0");
    require(
        token.balanceOf(address(this)) >= _amount,
        "Insufficient token balance"
    );
    token.safeTransfer(_to, _amount);
    emit AdminWithdrawal(_amount, _to);
}
```

The contract does not implement a receive() or fallback() function, nor does it have any mechanism to withdraw tokens other than the configured ERC20 token. Any ETH or other tokens sent to this contract would be permanently locked, with no way to retrieve them.

Single owner for admin withdrawals creates a centralization risk

Low Risk

The contract relies on a single owner for administrative functions, particularly for withdrawing collected tokens. If the owner's private key is lost or compromised, there would be no way to withdraw the collected tokens from the contract.

```
function adminWithdraw(
    uint256 _amount,
    address _to
) external nonReentrant onlyOwner {
    require(_to != address(0), "Invalid recipient address");
    require(_amount > 0, "Amount must be greater than 0");
    require(
        token.balanceOf(address(this)) >= _amount,
        "Insufficient token balance"
    );
    token.safeTransfer(_to, _amount);
    emit AdminWithdrawal(_amount, _to);
}
```

The reliance on a single owner address creates a single point of failure. If the owner's private key is lost or compromised, all funds collected in the contract would be permanently locked.

8 of 16 Findings

DepositThresholdNFT.sol

Support for deflationary (fee-on-transfer) tokens

Low Risk

The `mint` function assumes that the full `requiredAmount` of tokens is transferred, but if the ERC20 `token` charges transfer fees (deflationary token), the contract will receive less than expected and still mint the NFT. This allows users to underpay the intended deposit threshold.

```
function mint() external nonReentrant campaignActive {
    // ... state updates before interactions
    token.safeTransferFrom(msg.sender, address(this), requiredAmount); // may transfer less due to token fees
    _safeMint(msg.sender, tokenId);
}
```

Because the contract does not verify the actual received amount, an attacker can use a fee-on-transfer token to deposit fewer tokens than `requiredAmount` and still obtain an NFT, breaking the core economic assumptions of the campaign.

9 of 16 Findings

src/DepositThresholdNFT.sol

Non-Specific Solidity Pragma Version

Low Risk

The contract uses a floating pragma statement `pragma solidity ^0.8.20;` which allows the contract to be compiled with any compiler version starting from 0.8.20 up to (but not including) 0.9.0.

Floating pragmas can lead to inconsistent bytecode when compiled with different compiler versions, potentially introducing bugs or vulnerabilities that weren't present during testing. This is especially risky in production environments where consistent behavior is critical.

```
pragma solidity ^0.8.20;
```

It's recommended to lock the pragma to a specific version to ensure all deployments use exactly the same compiler version, guaranteeing consistent behavior across all environments.

PUSH0 Opcode Compatibility Issue

Low Risk

The contract uses Solidity version 0.8.20 which targets the Shanghai EVM version by default. This generates bytecode that includes the PUSH0 opcode.

```
pragma solidity ^0.8.20;
```

This can cause deployment failures on chains that haven't implemented the Shanghai upgrade or don't support the PUSH0 opcode, such as many Layer 2 solutions and alternative EVM-compatible blockchains.

To ensure compatibility across different chains, either:
1. Use an earlier Solidity version (< 0.8.20)
2. Explicitly specify the EVM version in your compiler settings (e.g., Paris instead of Shanghai)
3. Verify that your target deployment chain supports the PUSH0 opcode before deployment

Unnecessary Modifier Usage

Low Risk

The contract defines a `campaignActive()` modifier that appears to be unnecessary or inefficiently used.

```
modifier campaignActive() {
```

Unnecessary modifiers increase gas costs and code complexity without providing additional benefits. This can happen when:

1. The modifier is only used once (a direct check would be more efficient)
2. The modifier contains simple logic that could be inlined
3. The modifier duplicates functionality already present in other modifiers

Consider either removing this modifier if unused, inlining its logic if it's only used once, or refactoring to ensure it provides clear value in terms of code organization or reusability.

Loop contains `require` / `revert` statements

• Low Risk

The contract contains a loop with `require` or `revert` statements inside it:

```
for (uint256 i = 0; i < _dailyTokenAmounts.length; i++) {
```

This pattern is problematic because a single invalid item in the array will cause the entire transaction to fail, wasting gas and preventing the processing of valid items. This is especially concerning in functions that process user-provided arrays.

Instead of reverting on invalid items, consider:
1. Implementing a pattern that skips invalid items and continues processing
2. Returning an array of success/failure statuses for each item
3. Using a two-phase approach where validation happens before processing

This approach makes the contract more resilient and user-friendly, especially when handling batch operations.

No explicit check for ERC20 allowance before transfer

Info

In the `mint()` function, the contract doesn't check if the user has approved a sufficient allowance before attempting to transfer tokens:

```
token.safeTransferFrom(msg.sender, address(this), requiredAmount);
```

If the allowance is insufficient, the `safeTransferFrom` call will revert, but with a generic error message from the token contract. This creates a poor user experience, as users might not understand why their transaction failed. A more user-friendly approach would be to check the allowance first and provide a specific error message to guide the user.

Additional Insights: The lack of explicit allowance checking before token transfers creates a poor user experience but not a security vulnerability. When users attempt to mint without sufficient allowance, the transaction will revert with a generic error from the token contract rather than a clear, actionable message from the NFT contract. This makes troubleshooting difficult for users who may not understand why their transaction failed. The issue is purely related to user experience and error handling rather than contract security or fund safety. Adding an explicit allowance check with a custom error message would improve usability without changing the underlying security model.

Mixing of 0-based and 1-based indexing

Best Practices

The contract uses 1-based indexing for days (from the `getCurrentDay()` function) but 0-based indexing for arrays, which requires manual adjustments when accessing array elements:

```
// 1-based indexing for day
uint256 currentDay = getCurrentDay();

// Manual adjustment for 0-based array access
require(
    dailyWhitelistCount[currentDay] <
        dailyWhitelistLimits[currentDay - 1],
    "Daily whitelist limit reached"
);
```

While the current implementation correctly handles these adjustments, the mixed indexing approach increases cognitive load and creates potential for off-by-one errors in future code modifications. For consistency and to reduce the risk of future errors, it would be better to standardize on either 0-based or 1-based indexing throughout the contract.

Additional Insights: The mixed indexing approach (1-based for days, 0-based for arrays) creates unnecessary complexity and cognitive overhead. While the current implementation correctly handles the necessary adjustments, this design choice increases the risk of off-by-one errors in future modifications or by developers unfamiliar with the codebase. The issue is primarily related to code maintainability and developer experience rather than representing a current vulnerability. Standardizing on a single indexing approach throughout the contract would reduce this risk and improve code clarity without changing functionality.

Constructor does not validate initial owner address

Best Practices

The constructor sets the contract owner via `Ownable(_initialOwner)` without checking that `_initialOwner` is non-zero. If an inadvertent zero address is passed, ownership is assigned to the zero address, permanently disabling all `onlyOwner` functionality (including withdrawals).

```
constructor(
    uint256 _startingTimestamp,
    address _tokenAddress,
    uint256[] memory _dailyTokenAmounts,
    uint256[] memory _dailyWhitelistLimits,
    uint256 _campaignDuration,
    address _initialOwner
) ERC721("DepositThresholdNFT", "DTNFT") Ownable(_initialOwner) {
    // no check that _initialOwner != address(0)
    ...
}
```

Adding a requirement such as `require(_initialOwner != address(0), "Owner cannot be zero address");` would prevent accidental misconfiguration.

Explicit override missing for ERC721 safeTransferFrom overload

Best Practices

The contract overrides only the 4-parameter `safeTransferFrom` to revert, but does not explicitly override the 3-parameter overload. Although the 3-parameter version delegates to the 4-parameter override, explicitly overriding both overloads improves clarity and prevents any unintended transfer paths.

```
// Parent ERC721 implementation still exists in the contract:
function safeTransferFrom(address from, address to, uint256 tokenId) public
virtual override {
    safeTransferFrom(from, to, tokenId, "");
}
```

Adding an explicit override for the 3-parameter signature:

```
function safeTransferFrom(address from, address to, uint256 tokenId) public
pure override {
    revert("DepositThresholdNFT: Tokens are non-transferrable");
}
```

ensures that all transfer entry points are consistently blocked.

Disclaimer

Kindly note, the Audit Agent is currently in beta stage and no guarantee is being given as to the accuracy and/or completeness of any of the outputs the Audit Agent may generate, including without limitation this Report. The results set out in this Report may not be complete nor inclusive of all vulnerabilities. The Audit Agent is provided on an 'as is' basis, without warranties or conditions of any kind, either express or implied, including without limitation as to the outputs of the code scan and the security of any smart contract verified using the Audit Agent.

Blockchain technology remains under development and is subject to unknown risks and flaws. This Report does not indicate the endorsement of any particular project or team, nor guarantee its security. Neither you nor any third party should rely on this Report in any way, including for the purpose of making any decisions to buy or sell a product, service or any other asset.

To the fullest extent permitted by law, Nethermind disclaims any liability in connection with this Report, its content, and any related services and products and your use thereof, including, without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement. Nethermind does not warrant, endorse, guarantee, or assume responsibility for any product or service advertised or offered by a third party through the product, any open source or third-party software, code, libraries, materials, or information linked to, called by, referenced by or accessible through the report, its content, and the related services and products, any hyperlinked websites, any websites or mobile applications appearing on any advertising, and Nethermind will not be a party to or in any way be responsible for monitoring any transaction between you and any third-party providers of products or services. As with the purchase or use of a product or service through any medium or in any environment, you should use your best judgment and exercise caution where appropriate.

FOR AVOIDANCE OF DOUBT, THE REPORT, ITS CONTENT, ACCESS, AND/OR USAGE THEREOF, INCLUDING ANY ASSOCIATED SERVICES OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, INVESTMENT, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.