Security Review Report
NM-0449 Realio Districts

NETHERMIND SECURITY

(Mar 12, 2025)

Contents

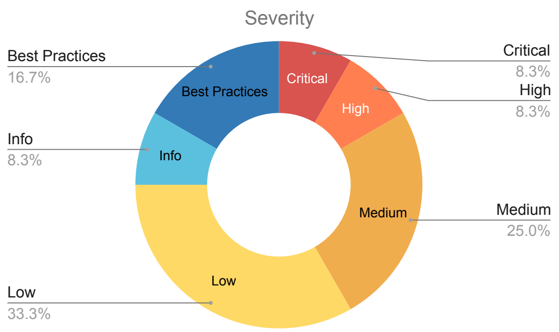
1 Executive Summary

This document presents the security review performed by Nethermind Security for Realio Districts smart contracts. Districts is a decentralized, open-source virtual world. Designed with creative professionals in mind, Districts offers a unique space for everyone to claim land and shape the future of virtual experience.

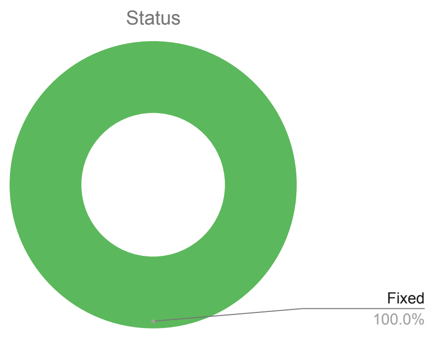
The reviewed contracts allow users to mint, buy and earn rewards from the LandPixel NFTs. The contracts also include a Marketplace where users can trade their NFTs through direct sells or auctions.

The audit comprises 1414 lines of solidity code. **The audit was performed using** (a) manual analysis of the codebase, (b) automated analysis tools, and (c) creation of test cases. **Along this document, we report** eleven points of attention, where one is classified as `Critical`, one is classified as `High`, three are classified as `Medium`, three are classified as `Low`, and three are classified as `Informational` or **Best Practices**. The issues are summarized in Fig. 1.

This document is organized as follows. Section 2 presents the files in the scope. Section 3 summarizes the issues. Section 4 presents the system overview. Section 5 discusses the risk rating methodology. Section 6 details the issues. Section 7 discusses the documentation provided by the client for this audit. Section 8 presents the compilation, tests, and automated tests. Section 9 concludes the document.



(a)



(b)

Fig. 1: Distribution of issues: Critical (1), **High** (1), **Medium** (3), **Low** (3), **Undetermined** (0), **Informational** (1), **Best Practices** (2).
Distribution of status: Fixed (11), **Acknowledged** (0), **Mitigated** (0), **Unresolved** (0)

Summary of the Audit

Audit Type	Security Review
Initial Report	February 26, 2025
Final Report	March 12, 2025
Repository	districts-smart-contracts
Commit	b0c0a6e73c535749d25911e78cd183294ff3418e
Final Commit	2f10821ad62a6d89760d05d38ed31a93319363c9
Documentation	README, Docs
Documentation Assessment	High
Test Suite Assessment	Medium

2 Audited Files

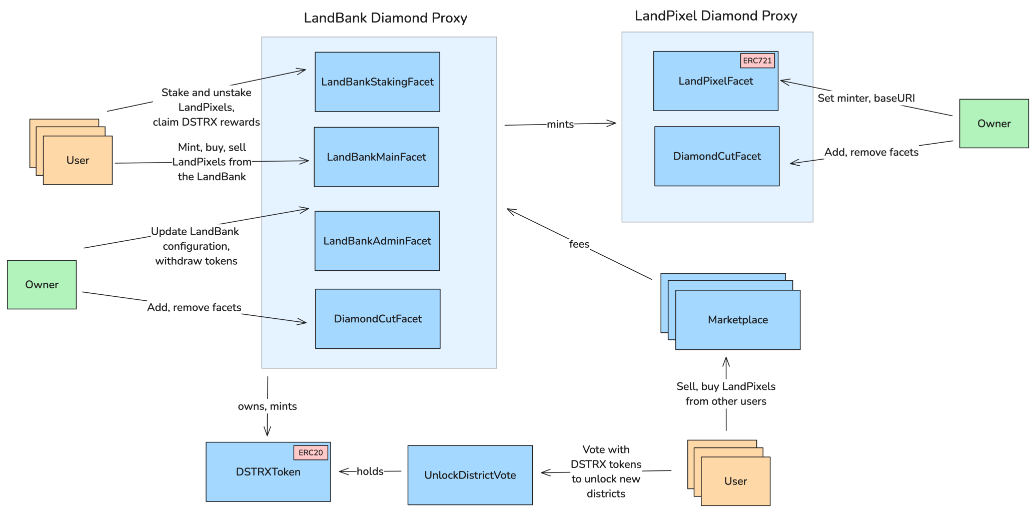
	Contract	LoC	Comments	Ratio	Blank	Total
1	contracts/DSTRXToken.sol	21	3	14.3%	5	29
2	contracts/Marketplace.sol	307	47	15.3%	71	425
3	contracts/Diamond.sol	43	21	48.8%	8	72
4	contracts/UnlockDistrictVote.sol	72	11	15.3%	26	109
5	contracts/facets/DiamondCutFacet.sol	9	13	144.4%	4	26
6	contracts/facets/OwnershipFacet.sol	21	1	4.8%	5	27
7	contracts/facets/LandBankMainFacet.sol	144	77	53.5%	59	280
8	contracts/facets/LandBankStakingFacet.sol	179	59	33.0%	69	307
9	contracts/facets/DiamondLoupeFacet.sol	98	44	44.9%	7	149
10	contracts/facets/LandPixelFacet.sol	87	6	6.9%	18	111
11	contracts/facets/LandBankAdminFacet.sol	66	39	59.1%	19	124
12	contracts/facets/AccessControlFacet.sol	5	1	20.0%	2	8
13	contracts/upgradeInitializers/LandPixelDiamondInit.sol	32	9	28.1%	7	48
14	contracts/upgradeInitializers/DiamondMultiInit.sol	13	9	69.2%	5	27
15	contracts/upgradeInitializers/LandBankDiamondInit.sol	59	11	18.6%	9	79
16	contracts/libraries/LibLandPixel.sol	14	1	7.1%	3	18
17	contracts/libraries/LibLandBank.sol	35	2	5.7%	6	43
18	contracts/libraries/LibDiamond.sol	179	17	9.5%	21	217
19	contracts/interfaces/IDiamondCut.sol	5	11	220.0%	3	19
20	contracts/interfaces/IDiamondLoupe.sol	11	20	181.8%	7	38
21	contracts/interfaces/IDiamond.sol	14	6	42.9%	4	24
	Total	**1414**	**408**	**28.9%**	**358**	**2180**

3 Summary of Issues

	Finding	Severity	Update
1	Reentrancy in makeOffer function allows attackers to drain funds in Marketplace	Critical	Fixed
2	Incorrect Computation of User Rewards	High	Fixed
3	`buyNow(...)` function does not check if the listing has expired	Medium	Fixed
4	`buyNow` function can be front-run by sellers	Medium	Fixed
5	acceptOffer function can be front-run	Medium	Fixed
6	Users can bid with the same amount if `highestBid` is too low	Low	Fixed
7	`buyNow(...)` function does not use the correct buy now price	Low	Fixed
8	`mintLandPixels` cannot be executed after reaching reward allocation	Low	Fixed
9	Marketplace receives zero fees if paid amount is too low	Info	Fixed
10	ReentrancyGuard uses default storage assignment	Best Practices	Fixed
11	`validPaymentToken` modifier should use whitelist to validate tokens	Best Practices	Fixed

4 System Overview

The **Districts** platform is a set of smart contracts, built on the **Realio Network**, EVM (Ethereum Virtual Machine). The project focuses on allowing users to claim scarce parcels of land within the ecosystem via the minting of “Land Pixels” NFTs, and to gain governance rights within the ecosystem via the mining of **DSTRX** governance tokens. Additionally, owners of "Land Pixels" NFTs can list and trade their assets on the **LandPixelMarketplace** contract, which is designed to automatically direct collected fees to the “Land Bank” treasury.



4.1 LandBank Diamond

The Land Bank is an ERC-2535 Diamond contract, which owns (and thus can mint) the **DSTRXToken** contract. It also controls the minting of **LandPixel** NFTs, allowing users to purchase digital land represented as NFTs with diminishing **DSTRX** token rewards. The system supports both direct minting of new pixels and trading of existing pixels through a floor price mechanism backed by the contract’s treasury. Users interact with **Land Bank** through these primary facets:

4.1.1 LandBankMainFacet

LandBankMainFacet contains most user-facing functions for interacting with the LandBank. It handles the minting, buying, and selling of LandPixels. The initial mint of Land Pixels distributes DSTRX as a reward. Main functions:

- **mintLandPixels**: Mints one or more LandPixel NFTs. Users must pay the total combined cost of LandPixels. DSTRX tokens get minted to user and LandBank per minted LandPixel.
- **buyLandPixels**: Buys one or more LandPixels from the bank. No DSTRX tokens are minted in this process.
- **sellLandPixel**: Sells a LandPixel to the LandBank at the floor price. Floor price is calculated as total RIO held in Land Bank divided by circulating supply of Land Pixel NFTs.

4.1.2 LandBankStakingFacet

LandBankStakingFacet contains user-facing staking-related code and methods. It handles the staking, unstaking of LandPixel NFTs and staking reward distribution. Stakers earn DSTRX tokens as a reward for LandPixel staking. To determine staking rewards, the contract employs an accumulator pattern, tracking rewards dynamically as they accrue over time. Main functions:

- **stakeLandPixel**: Stakes a LandPixel. Staking a LandPixel in the LandBank allows it to start earning DSTRX staking rewards. These rewards depend on the remaining supply of unminted DSTRX, starting at 1 DSTRX per block and decreasing over time, and are evenly distributed among all LandPixels currently staked.
- **unstakeLandPixel**: Remove a staked LandPixel from the LandBank and collect the accumulated DSTRX rewards earned over the entire staking period as pending rewards.
- **claimRewardForToken**: Claims rewards for a single staked LandPixel token.
- **claimAllRewards**: Claims all staking rewards for the calling user.

4.1.3 LandBankAdminFacet

LandBankAdminFacet includes the administrative functions for configuring and managing the LandBank settings. It also provides functionality to withdraw both native and ERC20 tokens from the contract. Main functions:

- **updateFeeRate**: Updates the fee rate charged for LandPixel buybacks.
- **updateRebuyDelay**: Updates the time delay before the LandBank's automatic buyback offer becomes active.
- **updatePixelCost**: Updates the base pixel cost.
- **updateMaxDistrictId**: Sets maximum district ID.
- **adminWithdraw**: Withdraw function for native token.
- **adminWithdrawTokens**: Withdraws miscellaneous ERC20 tokens sent to LandBank.

4.2 LandPixel Diamond

The **LandPixel** NFT contract is an Enumerable ERC721 ERC-2535 Diamond contract, where the **LandBank** serves as the only authorized minter. Each **LandPixel** is uniquely identified by a positive number or index, determined by the sequence in which Districts regions are released. The **tokenURI** links to the complete NFT metadata and must consistently follow OpenSea Metadata Standards; it is currently set to a fixed value. Users interact with **LandPixel** through the **LandPixelFacet** contract:

4.2.1 LandPixelFacet

LandPixelFacet implements the standard ERC721 interface with extra functions to enumerate minted NFTs. LandPixel NFTs owners can freely transfer their NFTs for use within the Districts ecosystem or in external applications like OpenSea. Main functions:

- **setBaseURI**: Sets the base token URI.
- **safeMint**: Mints a new LandPixel token using *_safeMint* function. It can only be called by the minter address.
- **setMinter**: Sets the minter address.

4.3 MarketPlace

LandPixelMarketplace contract supports listing and trading LandPixel NFTs, set up to automatically fund the LandBank treasury from collected fees. The collected fees in **native token** will increase the floor price of tokens in the LandBank contract. LandBank's logic remains independent of Marketplace specifics, allowing the deployment of multiple Marketplace contracts or instances, each configured to direct its fee revenue to the LandBank vault or treasury. Main functions:

- **listForSale**: Allows a seller to list a LandPixel NFT for sale.
- **unlist**: Allows a seller to deactivate a current listing.
- **bid**: Allows a bidder/buyer to bid on an existing listing.
- **makeOffer**: Allows a prospective buyer to make an offer on a LandPixel NFT even if there is no active listing of it in the Marketplace, which the owner can accept.
- **acceptOffer**: Allows the LandPixel owner to accept an offer that a buyer made with a *makeOffer* function call.
- **buyNow**: Allows a buyer to buy a listed LandPixel NFT that has a non-zero *buyItNow* price.
- **finalizeAuction**: Executable once an auction concludes, this function handles the necessary transfers and collects the applicable fees.
- **withdrawEscrow**: Allows bidders to withdraw any escrowed amount.
- **withdrawOffer**: Allows a bidder to withdraw their offer amount after the offer has expired.

4.4 UnlockDistrictVote

UnlockDistrictVote contract enables a token-based governance system to unlock districts in the LandBank ecosystem, using DSTRX tokens as voting power. It implements a threshold-based voting mechanism with partial token burning upon successful district unlocks.

- **vote**: Allows DSTRX token holders to cast votes for specific districts by locking their tokens in the contract.
- **withdrawVote**: Allows a seller to deactivate a current listing.
- **withdrawAfterUnlock**: Enables voters to retrieve their remaining tokens after a district is unlocked, with a portion burned according to the *burnPercentOnLockIn* rate.

4.5 Staking Rewards Mechanism

The **LandBank** staking system implements a diminishing rewards model based on the remaining token supply, with rewards distributed proportionally among staked **LandPixel** NFTs.

- The platform uses an accumulator pattern to track reward distribution, where each stake records a *rewardDebt* representing the global accumulated rewards at the time of staking or last claim.

- When users claim rewards, the system increases *stakingMintedRewards*, which reduces the remaining allocation and the reward rate per block for all stakers.
- If *stakingMintedRewards* reaches the *USER_TOTAL_SUPPLY* cap, pending rewards become unclaimable as the system prevents minting beyond the allocation limit, potentially leaving some rewards permanently unrealized in the platform.

5 Risk Rating Methodology

The risk rating methodology used by Nethermind Security follows the principles established by the OWASP Foundation. The severity of each finding is determined by two factors: **Likelihood** and **Impact**.

Likelihood measures how likely the finding is to be uncovered and exploited by an attacker. This factor will be one of the following values:

a) **High**: The issue is trivial to exploit and has no specific conditions that need to be met;

b) **Medium**: The issue is moderately complex and may have some conditions that need to be met;

c) **Low**: The issue is very complex and requires very specific conditions to be met.

When defining the likelihood of a finding, other factors are also considered. These can include but are not limited to motive, opportunity, exploit accessibility, ease of discovery, and ease of exploit.

Impact is a measure of the damage that may be caused if an attacker exploits the finding. This factor will be one of the following values:

a) **High**: The issue can cause significant damage, such as loss of funds or the protocol entering an unrecoverable state;

b) **Medium**: The issue can cause moderate damage, such as impacts that only affect a small group of users or only a particular part of the protocol;

c) **Low**: The issue can cause little to no damage, such as bugs that are easily recoverable or cause unexpected interactions that cause minor inconveniences.

When defining the impact of a finding, other factors are also considered. These can include but are not limited to Data/state integrity, loss of availability, financial loss, and reputation damage. After defining the likelihood and impact of an issue, the severity can be determined according to the table below.

		Severity Risk		
Impact	**High**	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Info/Best Practices	Low	Medium
	Undetermined	Undetermined	Undetermined	Undetermined
		Low	**Medium**	**High**
		Likelihood		

To address issues that do not fit a High/Medium/Low severity, Nethermind Security also uses three more finding severities: **Informational**, **Best Practices**, and **Undetermined**.

a) **Informational** findings do not pose any risk to the application, but they carry some information that the audit team intends to pass to the client formally;

b) **Best Practice** findings are used when some piece of code does not conform with smart contract development best practices;

c) **Undetermined** findings are used when we cannot predict the impact or likelihood of the issue.

6 Issues

6.1 [Critical] Reentrancy in makeOffer function allows attackers to drain funds in Marketplace

File(s): `contracts/Marketplace.sol`

Description: The `makeOffer` function in the `Marketplace.sol` contract is vulnerable to a reentrancy attack because it does not use a reentrancy guard (e.g., the nonReentrant modifier). The issue arises in the following scenario:

* When a user makes an offer (via `makeOffer`), the contract first checks if an existing offer is active. If so, it refunds the existing offer by adding its amount to `escrowBalances`.

* The function then calls an external contract via `IERC20(paymentToken).safeTransferFrom(msg.sender, address(this), amount)`.

* Since `paymentToken` is provided by the caller, an attacker could supply a malicious token contract that implements a reentrant fallback. When this external call is made, the malicious token could trigger a reentrant call to `makeOffer` before the state has been fully updated.

* By repeatedly reentering the function, the attacker can increase the `escrowBalances` for their address, enabling them to eventually withdraw more funds than they originally deposited and drain funds from the Marketplace.

The vulnerability specifically occurs because the state update (saving the new offer) happens after the external call, which violates the checks-effects-interactions pattern and leaves the contract exposed.

The problematic portion of the code is as follows:

```
    function makeOffer(
        uint256 tokenId,
        uint256 duration,
        address paymentToken,
        uint256 amount
    ) external payable validDuration(duration) validPaymentToken(paymentToken) {
        require(landPixelContract.ownerOf(tokenId) != msg.sender, "Cannot self-offer");

        // Check for existing offer from this user
        Offer storage existingOffer = offers[tokenId][msg.sender];
        if (existingOffer.active) {
            // Refund existing offer first
            if (existingOffer.paymentToken == address(0)) {
                escrowBalances[msg.sender][address(0)] += existingOffer.amount;
            } else {
                escrowBalances[msg.sender][existingOffer.paymentToken] += existingOffer.amount;
            }
        }

        // Handle payment
        if (paymentToken == address(0)) {
            require(msg.value == amount, "Incorrect ETH amount");
        } else {
            require(msg.value == 0, "ETH not accepted");
            require(amount > 0, "Amount must be > 0");
            //@audit Reentrant function call.
            IERC20(paymentToken).safeTransferFrom(msg.sender, address(this), amount);
        }

        // Create new offer
        offers[tokenId][msg.sender] = Offer({
            offerer: msg.sender,
            amount: amount,
            startTime: block.timestamp,
            duration: duration,
            active: true,
            paymentToken: paymentToken
        });

        emit OfferMade(tokenId, msg.sender, amount, paymentToken);
    }
```

Recommendation(s): Add the `nonReentrant` modifier (from OpenZeppelin's ReentrancyGuard) to the `makeOffer` function to prevent reentrant calls. For example:

```
    function makeOffer(
        uint256 tokenId,
        uint256 duration,
        address paymentToken,
        uint256 amount
    ) external payable nonReentrant validDuration(duration) validPaymentToken(paymentToken) {
        // ... remaining logic ...
    }
```

Status: Fixed.

Update from the client: Fixed in commit 6e78bb

6.2 [High] Incorrect Computation of User Rewards

File(s): `contracts/facets/LandBankMainFacet.sol`

Description: The contract tracks the cumulative rewards earned per `NFT` using the `accumulatedRewardsPerShare` variable. This variable is crucial for determining the rewards accrued by each user.

The expected behavior is that `accumulatedRewardsPerShare` should be updated whenever the per-block rewards change. The per-block rewards are determined by: a) The number of staked `NFTs`. b) The total rewards per block, which depends on `remainingAllocation`.

However, the contract fails to update `accumulatedRewardsPerShare` when `remainingAllocation` changes, leading to incorrect reward calculations. As a result, users may receive inaccurate rewards, potentially causing discrepancies in the reward distribution.

Recommendation: Ensure that `accumulatedRewardsPerShare` is updated whenever `remainingAllocation` or the number of staked NFTs changes to maintain accurate reward calculations.

Status: Fixed.

Update from the client: Fixed in commit cd21c6

6.3 [Medium] `buyNow(...)` function does not check if the listing has expired

File(s): `contracts/Marketplace.sol`

Description: The `buyNow(...)` function is used to purchase tokens listed with either:

- `SellType == FixedPrice`, or;
- `SellType == Auction` when `buyNowPrice != 0`;

Each listing has a defined lifespan, determined by its duration and the timestamp when it was created. However, the `buyNow(...)` function does not check whether the listing has expired.

This can lead to two unintended behaviors:

1. **Fixed-Price listings:** Users can still purchase a listing even after its valid duration has passed;
2. **Auction listings:** Users can directly buy a token using `buyNowPrice`, even if the auction has ended, as long as `finalizeAuction(...)` has not been called;

Recommendation(s) Consider validating the listing's lifespan within the `buyNow(...)` function to ensure purchases are only allowed if the listing is still active.

Status: Fixed.

Update from the Client Fixed in commit 0e72ee.

6.4 [Medium] buyNow function can be front-run by sellers

File(s): contracts/Marketplace.sol

Description: When an NFT marketplace listing is placed, the seller can optionally set a buy now price. For SaleType.Auction sales, the buyNowPrice parameter is used as the buy now price. For SaleType.FixedPrice sales, highestBid parameter is used as the buy now price.

```
    function listForSale(
        uint256 tokenId,
        uint256 startingPrice,
        SaleType saleType,
        uint256 duration,
        uint256 buyNowPrice,
        address paymentToken
    ) external validDuration(duration) validPaymentToken(paymentToken) {
    //... REDACTED FOR BREVITY ...
        listings[tokenId] = Listing({
            saleType: saleType,
            seller: msg.sender,
            highestBidder: msg.sender,
            highestBid: startingPrice,
            startTime: block.timestamp,
            duration: duration,
            active: true,
            // Buy now price is set here
            buyNowPrice: buyNowPrice,
            paymentToken: paymentToken,
            escrowedAmount: 0
        });
```

By calling the buyNow function with the corresponding tokenId, buyer can purchase the NFT token immediately without waiting for auction to happen.

```

    function buyNow(uint256 tokenId) external payable nonReentrant {
        Listing storage listing = listings[tokenId];
        require(msg.sender != listing.seller, "Cannot buy own listing");
        require(listing.active, "Not for sale");
//... REDACTED FOR BREVITY ...
}

```

Notice that only tokenId is taken as a function argument in the buyNow function. Since the buyers only specify the tokenId they are purchasing, this can lead to front-running attacks where the seller can change the buy now price before the purchase is made. This is possible because sellers can unlist the NFT and re-list again with different buy now prices. This is only possible for SaleType.FixedPrice type sales as only those types of listing are allowed to be unlisted. Here's a sample attack scenario:

1. Seller creates a market listing using the listForSale function setting sale type -> SaleType.FixedPrice;
2. Buyer notices the listing and calls buyNow function with the tokenId in the first step;
3. Seller front-runs the buyNow transaction and first calls the unlist function and then calls listForSale function with different buy now prices;
4. Buyer's buyNow transaction goes through and purchase is made for a much higher price than buyer intended;

Recommendation(s): Each listing created via the listForSale function should have a unique id. After a listing is created, they should be immutable. Users should use the unique listing ids to interact with the protocol.

Status: Fixed.

Update from the client: Fixed in commit a50086.

6.5 [Medium] acceptOffer function can be front-run

File(s): contracts/Marketplace.sol

Description: In the `Marketplace` contract, Landpixel NFT owners can auction their NFTs or accept offers made to their NFTs. If an offer is made to a Landpixel NFT, the NFT owner will call the `acceptOffer(uint256 tokenId, address offerer)` function to accept the offer.

```
    function acceptOffer(uint256 tokenId, address offerer) external nonReentrant {
        require(landPixelContract.ownerOf(tokenId) == msg.sender, "Not the token owner");

        Offer storage offer = offers[tokenId][offerer];
        require(offer.active, "Offer inactive");
        require(block.timestamp < offer.startTime + offer.duration, "Offer expired");

        uint256 amount = offer.amount;
        address paymentToken = offer.paymentToken;

        // Deactivate the accepted offer
        offer.active = false;

        // Calculate fees
        uint256 feeAmount = (amount * marketplaceFee) / 10000;
        uint256 sellerAmount = amount - feeAmount;

        // Transfer NFT
        landPixelContract.transferFrom(msg.sender, offerer, tokenId);

        // Transfer payment
        if (paymentToken == address(0)) {
            (bool success, ) = payable(msg.sender).call{value: sellerAmount}("");
            require(success, "Seller payment failed");
            (bool feeSuccess, ) = payable(landBank).call{value: feeAmount}("");
            require(feeSuccess, "Fee transfer failed");
        } else {
            IERC20(paymentToken).safeTransfer(msg.sender, sellerAmount);
            IERC20(paymentToken).safeTransfer(landBank, feeAmount);
        }

        emit SaleFinalized(tokenId, offerer, amount, paymentToken);
    }
```

However, the contract doesn't account for the fact that offers can be changed after they're made. This means a malicious buyer could change their offer right before the NFT owner accepts it, potentially lowering the offer after the owner has already agreed to the original terms. Imagine the following attack scenario:

1. Attacker makes an offer to a Landpixel NFT via `makeOffer(...)` function;
2. NFT owner accepts the offer by calling `acceptOffer(...)` and publishes tx onchain;
3. Attacker front-runs the NFT owner's transaction and updates the offer with `makeOffer(...)` function;
4. NFT owner's `acceptOffer(...)` is executed and owner sells their NFT for less than agreed amount;

Recommendation(s): Track offers by storing unique identifiers. Each offer should have a unique identifier so the NFT owner can accept without taking a risk.

Status: Fixed.

Update from the client: Fixed in commit 49bdd5.

6.6 [Low] Users can bid with the same amount if `highestBid` is too low

File(s): `contracts/Marketplace.sol`

Description: In the `Marketplace` contract, users can place bids on listings with `SaleType == Auction`. The contract enforces that each new bid must exceed the previous `highestBid` by a percentage of it, using the following calculation:

```
...
uint256 minBid = listing.highestBid + ((listing.highestBid * MIN_BID_INCREMENT_BPS) / 10000);
require(bidAmount >= minBid, "Bid too low");
...
```

However, if `listing.highestBid * MIN_BID_INCREMENT_BPS` results in a value lower than `10000`, the division by `10000` will round down to zero (in integer division). As a result, `minBid` will be equal to `highestBid`, allowing users to place bids at the same price.

This behavior can lead to auction sniping, where a bidder submits a last-minute bid at the same price as the current highest bid, potentially undermining the intended competitive bidding process.

Recommendation(s): Consider modifying the bid enforcement logic to ensure `minBid` is always strictly greater than `highestBid`. Possible solutions include:

- Setting a minimum price to prevent `minBid` from equaling `highestBid`;
- Explicitly enforcing `bidAmount > highestBid` in the require statement;

Status: Fixed.

Update from the client: Fixed in commit 07ee7c.

Update from Nethermind: This wrongly enforces the `bidAmount` to be strictly bigger than `minBid` always. `bidAmount` should be strictly higher than previous bid, but greater or equal to `minBid`

Update from the client: Fixed in commit 91bb48.

6.7 [Low] `buyNow(...)` function does not use the correct buy now price

File(s): `contracts/Marketplace.sol`

Description: In the `buyNow(...)` function, buy now price is different depending on the `SaleType`:

```
uint256 price = listing.saleType == SaleType.FixedPrice ? listing.highestBid : listing.buyNowPrice;
```

- For the `SaleType.FixedPrice` , `highestBid` is used as the price;
- For the `SaleType.Auction`, `buyNowPrice` is used as the price;

This can lead to ambiguity for the users and they might assume the incorrect buy now price is valid. A seller can put a very high `highestBid` price and a very low `buyNowPrice` for a fixed price sale. The users might assume the `buyNowPrice` is what they'll pay. However, they'll pay the `highestBid` price and will pay more than they intended.

Recommendation(s): Only use the `buyNowPrice` in the buyNow function.

Status: Fixed.

Update from the Client Fixed in commit e13dc9.

6.8 [Low] `mintLandPixels` cannot be executed after reaching reward allocation

File(s): `contracts/facets/LandBankMainFacet.sol`

Description: The `mintLandPixels(...)` function allows users to mint new `NFTs` by paying the `pixelCost` per NFT. In return, users receive both the `NFT` and `DSTRX` rewards.

However, if minting a new `NFT` would cause the total user rewards allocation to be exceeded, the function reverts, preventing any further minting. This makes it impossible to mint new `NFTs` even if users are willing to forgo rewards.

Recommendation: Modify the function logic to allow minting even when the user rewards allocation is exhausted. Instead of reverting, conditionally distribute rewards based on the remaining allocation.

Status: Fixed.

Update from the client: Fixed in commit 9c6eff.

6.9 [Info] Marketplace receives zero fees if paid amount is too low

File(s): `contract/Marketplace.sol`

Description: The `Marketplace` contract gets a fee from every sell done through it and send it to the `LandBank` contract.

```
// Calculate fees
// @audit - Fee computation rounds down
uint256 feeAmount = (amount * marketplaceFee) / 10000;
uint256 sellerAmount = amount - feeAmount;
```

If the price of the sell is to low, the computation of the fees will round towards zero making the fee amount zero.

Recommendation(s): Consider rounding up if fees must be enforced.

Status: Fixed.

Update from the client: Fixed in commit 0cbb98.

6.10 [Best Practice] ReentrancyGuard uses default storage assignment

File(s): `contracts/facets/LandBankMainFacet.sol`, `contracts/facets/LandBankStakingFacet.sol`

Description: The protocol follows the Diamond Pattern, where storage slots are managed using a structured domain-based algorithm to prevent conflicts. However, the `ReentrancyGuard` contract does not follow this pattern and is instead inherited in two facets:

- `LandBankMainFacet`;
- `LandBankStakingFacet`;

Since `ReentrancyGuard` relies on Solidity's default storage assignment, it introduces a risk of storage collisions in future upgrades, especially as new facets are added or modified.

Recommendation: Adapt the `ReentrancyGuard` contract to use the same structured slot assignment as the rest of the protocol. The upgradeable version of `ReentrancyGuard` already follows this pattern and can be used as a reference.

Status: Fixed.

Update from the client: Fixed in commit 3e8057.

6.11 [Best Practice] `validPaymentToken` modifier should use whitelist to validate tokens

File(s): `contracts/Marketplace`

Description: While creating a listing using `listForSale` or making a offer using `makeOffer`, the users are allowed to set `paymentToken` to any address as long as the address is a smart contract. Later in those functions, external calls are made to the `paymentToken` address to interact with the tokens. While this approach offers freedom to the users to choose any token they want, it also opens a significant attack surface to the attackers. One of the attacks that this issue can lead to is already explained in the `Reentrancy in makeOffer function allows attackers to drain funds in Marketplace` issue.

One of the potential issues that this can lead to could be setting `paymentToken` address to a LandPixel NFT address in the `listForSale` function. If the bidders are LandPixel NFT owners and if they set the `bidAmount` to an NFT id that they own, they can inadvertently transfer their NFTs to the Marketplace contract with no way to get that back. Relevant code piece:

```
function bid(uint256 tokenId, uint256 bidAmount) external payable nonReentrant {
// REDACTED FOR BREVITY
 // Handle payments first (CEI pattern)
        if (listing.paymentToken == address(0)) {
            require(msg.value == bidAmount, "Incorrect ETH amount");
        } else {
            require(msg.value == 0, "ETH not accepted");
            // Transfer tokens to escrow
            //@audit Here the paymentToken can be LandPixel NFT and not ERC20. This would cause NFT to be locked in the
            ↪  contract.
            IERC20(listing.paymentToken).safeTransferFrom(msg.sender, address(this), bidAmount);
        }
```

Another potential issue is using fee-on-transfer tokens as `paymentToken`. In that scenario, the fee-on-transfer token would incur fees during the transfer function. This could lead to miscalculation of actual token amounts held in the Marketplace contract. These type of tokens shouldn't be allowed to be used in the Marketplace.

Recommendation(s): We recommend adding a token whitelisting logic. Only allowed tokens should be used in the Marketplace.

Status: Fixed.

Update from the client: Fixed in commit c817f9.

7 Documentation Evaluation

Software documentation refers to the written or visual information that describes the functionality, architecture, design, and implementation of software. It provides a comprehensive overview of the software system and helps users, developers, and stakeholders understand how the software works, how to use it, and how to maintain it. Software documentation can take different forms, such as user manuals, system manuals, technical specifications, requirements documents, design documents, and code comments. Software documentation is critical in software development, enabling effective communication between developers, testers, users, and other stakeholders. It helps to ensure that everyone involved in the development process has a shared understanding of the software system and its functionality. Moreover, software documentation can improve software maintenance by providing a clear and complete understanding of the software system, making it easier for developers to maintain, modify, and update the software over time. Smart contracts can use various types of software documentation. Some of the most common types include:

- Technical whitepaper: A technical whitepaper is a comprehensive document describing the smart contract's design and technical details. It includes information about the purpose of the contract, its architecture, its components, and how they interact with each other;
- User manual: A user manual is a document that provides information about how to use the smart contract. It includes step-by-step instructions on how to perform various tasks and explains the different features and functionalities of the contract;
- Code documentation: Code documentation is a document that provides details about the code of the smart contract. It includes information about the functions, variables, and classes used in the code, as well as explanations of how they work;
- API documentation: API documentation is a document that provides information about the API (Application Programming Interface) of the smart contract. It includes details about the methods, parameters, and responses that can be used to interact with the contract;
- Testing documentation: Testing documentation is a document that provides information about how the smart contract was tested. It includes details about the test cases that were used, the results of the tests, and any issues that were identified during testing;
- Audit documentation: Audit documentation includes reports, notes, and other materials related to the security audit of the smart contract. This type of documentation is critical in ensuring that the smart contract is secure and free from vulnerabilities.

These types of documentation are essential for smart contract development and maintenance. They help ensure that the contract is properly designed, implemented, and tested, and they provide a reference for developers who need to modify or maintain the contract in the future.

Remarks about the Realio Districts documentation

The Realio Districts protocol has documentation explaining its product. Beyond public documentation, they have provided a Readme file explaining the architecture of the multiple contracts and the responsibilities of each of them. In addition to this, the team was available to answer any questions raised during the engagement.

8 Complementary Checks

8.1 Compilation Output

```
> npx hardhat compile
Generating typings for: 96 artifacts in dir: typechain-types for target: ethers-v6
Successfully generated 244 typings!
Compiled 92 Solidity files successfully (evm target: paris).
```

8.2 Tests Output

```
> npx hardhat test --network hardhat


  LandBank
    Deployment
       Should set the right owner
       Should deploy LandPixel and DSTRXToken contracts
       Should set initial values correctly
    Claiming Land
       Should allow minting land with sufficient payment
       Should not allow minting land with insufficient payment
       Should not allow minting land beyond maxDistrictId
       Should refund excess payment when minting land
    Buying Land from LandBank
       Should allow buying multiple LandPixels from the bank in a single transaction
    Selling Land to Bank
       Should not allow selling land back to the bank before rebuyDelay seconds have elapsed
       Should allow selling land back to the bank after rebuyDelay seconds have elapsed
       Should not allow selling land not owned by the caller
    Admin Functions
       Should allow owner to update fee rate
       Should allow owner to update pixel cost
       Should allow owner to update max district ID
       Should not allow non-owners to update contract parameters
       Should allow admins adjustments to rebuyDelay to affect LandBank rebuy wait period
    Withdrawals
       Should allow owner to withdraw native tokens
       Should allow owner to withdraw ERC20 tokens
       Should not allow non-owners to withdraw
    Staking
       Should allow staking a LandPixel
       Should allow staking multiple LandPixels
       Should not allow staking a LandPixel that is not owned
       Should allow claiming staking rewards
       Should not allow claiming rewards for unstaked LandPixel
       Should calculate rewards correctly based on blocks elapsed
       Should respect the staking allocation limit
    Buying Land from Bank
       Should allow buying unstaked land from the bank
       Should not allow buying staked land from the bank
       Should allow buying previously staked but now unstaked land from the bank
       Should not allow buying land not owned by the bank
       Should refund excess payment when buying land from bank
    Floor Price
       Should calculate the correct floor price
       Should update floor price when contract balance changes
       Should accept direct payments from arbitrary payers and increase floor price
       Should calculate the correct floor price based on circulating supply
    Reentrancy Protection
       Should prevent reentrant calls during mintLandPixels

  LandPixel
     Creates a token collection with a name
     Creates a token collection with a symbol
     Is able to query the NFT balances of an address
     Is able to mint new NFTs to the collection to collector
     Prevents NFTs from being re-minted when they have already been minted
     Emits a transfer event for newly minted NFTs
     Emits a Transfer event when transferring a NFT
     Approves an operator wallet to spend owner NFT
     Emits an Approval event when an operator is approved to spend a NFT
     Allows operator to transfer NFT on behalf of owner
     Approves an operator to spend all of an owner's NFTs
     Emits an ApprovalForAll event when an operator is approved to spend all NFTs
     Removes an operator from spending all of owner's NFTs
     Allows operator to transfer all NFTs on behalf of owner
     Only allows contractOwner to mint NFTs


  51 passing (6s)
```

```
> forge install foundry-rs/forge-std --no-commit
> forge test
[] Compiling...
No files changed, compilation skipped

Ran 18 tests for test/DSTRXToken.t.sol:DSTRXTokenTest
[PASS] test_allowance() (gas: 15190)
[PASS] test_approve() (gas: 43661)
[PASS] test_approveRevertInvalidApprover() (gas: 11742)
[PASS] test_approveRevertInvalidSpender() (gas: 11769)
[PASS] test_balanceOfAddress0() (gas: 10798)
[PASS] test_balanceOfAddressSupplyOwner() (gas: 12878)
[PASS] test_decimals() (gas: 8511)
[PASS] test_name() (gas: 12555)
[PASS] test_symbol() (gas: 12530)
[PASS] test_totalSupply() (gas: 10554)
[PASS] test_transfer() (gas: 50946)
[PASS] test_transferFrom() (gas: 81832)
[PASS] test_transferFromRevertInsufficientAllowance() (gas: 42745)
[PASS] test_transferFromRevertInsufficientAllowanceFor0x0() (gas: 16580)
[PASS] test_transferFromRevertInvalidReceiver() (gas: 28428)
[PASS] test_transferRevertInsufficientBalance() (gas: 16626)
[PASS] test_transferRevertInvalidReceiver() (gas: 11805)
[PASS] test_transferRevertInvalidSender() (gas: 11780)
Suite result: ok. 18 passed; 0 failed; 0 skipped; finished in 14.99ms (34.30ms CPU time)

Ran 27 tests for test/LandBank.t.sol:LandBankTest
[PASS] testAccessControl() (gas: 84917)
[PASS] testAdminFacetUpdateFeeRate() (gas: 44075)
[PASS] testAdminFacetUpdateMaxDistrictId() (gas: 61295)
[PASS] testAdminFacetUpdatePixelCost() (gas: 44173)
[PASS] testDeployment() (gas: 48611)
[PASS] testFailStakingEdgeCases() (gas: 86911)
[PASS] testFuzzMintLandPixels(uint256[]) (runs: 256, : 9355796, ~: 10485316)
[PASS] testGetCurrentMintReward() (gas: 1062238)
[PASS] testGettersAndSetters() (gas: 48986)
[PASS] testMintLandBeyondMaxDistrictId() (gas: 89586)
[PASS] testMintLandPixelsWithDuplicateTokenIds() (gas: 290710)
[PASS] testMintLandPixelsWithExcessPayment() (gas: 501483)
[PASS] testMintLandPixelsWithInvalidPaymentAmount() (gas: 86921)
[PASS] testMintLandPixelsWithMultipleTokens() (gas: 785294)
[PASS] testMintLandWithInsufficientPayment() (gas: 84531)
[PASS] testMintLandWithSufficientPayment() (gas: 494621)
[PASS] testMultipleStakersRewardDistribution() (gas: 1029734)
[PASS] testReceiveFunction() (gas: 19312)
[PASS] testRewardCalculation() (gas: 676600)
[PASS] testRewardClaimingAndSupplyLimit() (gas: 759116)
[PASS] testSellLandPixelBeforeRebuyDelay() (gas: 525253)
[PASS] testSellLandPixelNotOwner() (gas: 494876)
[PASS] testStaking() (gas: 675147)
[PASS] testStakingRewardCalculation() (gas: 681066)
[PASS] testStakingStorageConsistency() (gas: 685468)
[PASS] testStakingStorageInitialization() (gas: 51691)
[PASS] testUnstaking() (gas: 666024)
Suite result: ok. 27 passed; 0 failed; 0 skipped; finished in 377.75ms (373.00ms CPU time)

Ran 2 test suites in 386.96ms (392.74ms CPU time): 45 tests passed, 0 failed, 0 skipped (45 total tests)
```

8.3 Automated Tools

8.3.1 AuditAgent

All the relevant issues raised by the AuditAgent have been incorporated into this report. The AuditAgent is an AI-powered smart contract auditing tool that analyses code, detects vulnerabilities, and provides actionable fixes. It accelerates the security analysis process, complementing human expertise with advanced AI models to deliver efficient and comprehensive smart contract audits. Available at https://app.auditagent.nethermind.io.

9 About Nethermind

Nethermind is a Blockchain Research and Software Engineering company. Our work touches every part of the web3 ecosystem - from layer 1 and layer 2 engineering, cryptography research, and security to application-layer protocol development. We offer strategic support to our institutional and enterprise partners across the blockchain, digital assets, and DeFi sectors, guiding them through all stages of the research and development process, from initial concepts to successful implementation.

We offer security audits of projects built on EVM-compatible chains and Starknet. We are active builders of the Starknet ecosystem, delivering a node implementation, a block explorer, a Solidity-to-Cairo transpiler, and formal verification tooling. Nethermind also provides strategic support to our institutional and enterprise partners in blockchain, digital assets, and decentralized finance (DeFi). In the next paragraphs, we introduce the company in more detail.

Blockchain Security: At Nethermind, we believe security is vital to the health and longevity of the entire Web3 ecosystem. We provide security services related to Smart Contract Audits, Formal Verification, and Real-Time Monitoring. Our Security Team comprises blockchain security experts in each field, often collaborating to produce comprehensive and robust security solutions. The team has a strong academic background, can apply state-of-the-art techniques, and is experienced in analyzing cutting-edge Solidity and Cairo smart contracts, such as ArgentX and StarkGate (the bridge connecting Ethereum and StarkNet). Most team members hold a Ph.D. degree and actively participate in the research community, accounting for 240+ articles published and 1,450+ citations in Google Scholar. The security team adopts customer-oriented and interactive processes where clients are involved in all stages of the work.

Blockchain Core Development: Our core engineering team, consisting of over 20 developers, maintains, improves, and upgrades our flagship product - the Nethermind Ethereum Execution Client. The client has been successfully operating for several years, supporting both the Ethereum Mainnet and its testnets, and now accounts for nearly a quarter of all synced Mainnet nodes. Our unwavering commitment to Ethereum's growth and stability extends to sidechains and layer 2 solutions. Notably, we were the sole execution layer client to facilitate Gnosis Chain's Merge, transitioning from Aura to Proof of Stake (PoS), and we are actively developing a full-node client to bolster Starknet's decentralization efforts. Our core team equips partners with tools for seamless node set-up, using generated docker-compose scripts tailored to their chosen execution client and preferred configurations for various network types.

DevOps and Infrastructure Management: Our infrastructure team ensures our partners' systems operate securely, reliably, and efficiently. We provide infrastructure design, deployment, monitoring, maintenance, and troubleshooting support, allowing you to focus on your core business operations. Boasting extensive expertise in Blockchain as a Service, private blockchain implementations, and node management, our infrastructure and DevOps engineers are proficient with major cloud solution providers and can host applications in-house or on clients' premises. Our global in-house SRE teams offer 24/7 monitoring and alerts for both infrastructure and application levels. We manage over 5,000 public and private validators and maintain nodes on major public blockchains such as Polygon, Gnosis, Solana, Cosmos, Near, Avalanche, Polkadot, Aptos, and StarkWare L2. Sedge is an open-source tool developed by our infrastructure experts, designed to simplify the complex process of setting up a proof-of-stake (PoS) network or chain validator. Sedge generates docker-compose scripts for the entire validator set-up based on the chosen client, making the process easier and quicker while following best practices to avoid downtime and being slashed.

Cryptography Research: At Nethermind, our Cryptography Research team is dedicated to continuous internal research while fostering close collaboration with external partners. The team has expertise across a wide range of domains, including cryptography protocols, consensus design, decentralized identity, verifiable credentials, Sybil resistance, oracles, and credentials, distributed validator technology (DVT), and Zero-knowledge proofs. This diverse skill set, combined with strong collaboration between our engineering teams, enables us to deliver cutting-edge solutions to our partners and clients.

Smart Contract Development & DeFi Research: Our smart contract development and DeFi research team comprises 40+ world-class engineers who collaborate closely with partners to identify needs and work on value-adding projects. The team specializes in Solidity and Cairo development, architecture design, and DeFi solutions, including DEXs, AMMs, structured products, derivatives, and money market protocols, as well as ERC20, 721, and 1155 token design. Our research and data analytics focuses on three key areas: technical due diligence, market research, and DeFi research. Utilizing a data-driven approach, we offer in-depth insights and outlooks on various industry themes.

Our suite of L2 tooling: Warp is Starknet's approach to EVM compatibility. It allows developers to take their Solidity smart contracts and transpile them to Cairo, Starknet's smart contract language. In the short time since its inception, the project has accomplished many achievements, including successfully transpiling Uniswap v3 onto Starknet using Warp.

- **Voyager** is a user-friendly Starknet block explorer that offers comprehensive insights into the Starknet network. With its intuitive interface and powerful features, Voyager allows users to easily search for and examine transactions, addresses, and contract details. As an essential tool for navigating the Starknet ecosystem, Voyager is the go-to solution for users seeking in-depth information and analysis;

- **Horus** is an open-source formal verification tool for StarkNet smart contracts. It simplifies the process of formally verifying Starknet smart contracts, allowing developers to express various assertions about the behavior of their code using a simple assertion language;

- **Juno** is a full-node client implementation for Starknet, drawing on the expertise gained from developing the Nethermind Client. Written in Golang and open-sourced from the outset, Juno verifies the validity of the data received from Starknet by comparing it to proofs retrieved from Ethereum, thus maintaining the integrity and security of the entire ecosystem.

Learn more about us at nethermind.io.

General Advisory to Clients

As auditors, we recommend that any changes or updates made to the audited codebase undergo a re-audit or security review to address potential vulnerabilities or risks introduced by the modifications. By conducting a re-audit or security review of the modified codebase, you can significantly enhance the overall security of your system and reduce the likelihood of exploitation. However, we do not possess the authority or right to impose obligations or restrictions on our clients regarding codebase updates, modifications, or subsequent audits. Accordingly, the decision to seek a re-audit or security review lies solely with you.

Disclaimer