Le7el Network
Investor NFT
Smart Contract Review

coinspect

L7EL

Investor NFT Contracts

Smart Contract Review

Version: v240212 Prepared for: Le7el Network February 2024

Smart Contract Review

1. Executive Summary

Le7el introduces a decentralized platform that allows for different contributors to create games and support the system's life-cycle. The system is monetized through game sales to players, and rewards to developers and maintainers of the network.

The $L7L token is the central element of the LE7EL network's tokenomics. It serves as a means to incentivize and distribute value fairly among different stakeholders in the network. One of the stakeholders of the ecosystem is the Investor, who represent 10% of the total network supply.

In **February 2024**, **Le7el** engaged **Coinspect** to perform a source code review of its Investor NFT smart contracts. The objective of the project was to evaluate the security of the NFT token and the minting mechanisms for the Investor in the Le7el network.

Solved	Caution Advised	Resolution Pending
High	High	High
0	0	0
Medium	Medium	Medium
1	0	0
Low	Low	Low
3	0	0
No Risk	No Risk	No Risk
4	1	0
Total	Total	Total
8	**1**	**0**

L7L-1 shows how different whitelisted users for distinct NFTs are not able to mint any token. L7L-2 describes a risk of denial of service associated with the beneficiary account

and the minting process. L7L-3 is related to the deployment process and how whitelisted users might waste gas when trying to mint an NFT on this stage. L7L-4 shows how the owner of the minter contract can steal user's payments without minting any token in exchange.

2. Summary of Findings

2.2 Findings where caution is advised

These issues have been addressed, but their risks have not been fully mitigated. Any future changes to the codebase should be carefully evaluated to avoid exacerbating these issues or increasing their probability.

Findings with a risk of None pose no threat, but document an implicit assumption which must be taken into account. Once acknowledged, these are considered solved.

Id	Title	Risk
L7L-8	Ownership can be mistakenly lost	None

2.3 Solved issues & recommendations

These issues have been fully fixed or represent recommendations that could improve the long-term security posture of the project.

Id	Title	Risk
L7L-1	Whitelisted investors for new NFTs are not able to mint any token	Medium
L7L-2	Malicious beneficiary can prevent anyone from minting	Low
L7L-3	Users will waste gas when trying to mint after the investor deployment	Low
L7L-4	The owner can steal payments without minting any token	Low

L7L-5	Users can unsuspectingly mint the wrong token	None
L7L-6	Users can precalculate the NFT ID as the semi-random generation has no randomness	None
L7L-7	Minting phases can be replaced by an enum	None
L7L-9	Different instances of the Investor NFT will have the same name and symbol	None

3. Scope

The scope was set to be the repository at https://gitlab.com/le7el/play/crs-nfts at commit `14f65f315b8fd49d39a129541b90cc19639b046e`. Specifically, the contracts in scope are:

- `InvestorNFTV1.sol`
- `InvestorNFTMinterV1.sol`
- Unit tests

Coinspect used Le7el Network's whitepaper to determine the economics and business model in question. A threat model has been created, guided by the whitepaper altogether with the codebase review.

4 Assessment

The Investor Token is an NFT which allows investors to participate on a game platform. This tokenization process is a way where investors get an on-chain representation of their shares over the game.

This on-chain ownership title is represented by the `InvestorNFTV1`, a **ERC721** token that will be deployed on the Ethereum chain.

The minting process for each NFT token will be coordinated by a Minting contract, which is deployed in advance. When deploying the NFT, the address of the minter contract is provided and no one but this account has the privilege to mint a token from the NFT contract.

The minter is implemented in the `InvestorNFTMinterV1` contract and is in charge of coordinating different minting phases. Tokens can be minted by investors in three different phases, GTD (whitelist), FCFS (whitelist), and Public (open). For the first two phases, a merkle root for a tree comprised by the whitelisted investors is provided into the minter contract. Each whitelisted phase will have two separate merkle roots. These two phases require investors to provide a valid proof along with their position on the whitelist. In latter minting phase, anyone can mint, as long as there is enough supply (capped by the NFT contract at `2222` tokens).

When minting an NFT, it is checked that the recipient is either an external owned account (EOA) or a contract implementing the `ERC721TokenReceiver` to mitigate the risk of irreversibly minting a token to an address that is not capable of handling it.

4.1 Security assumptions

The following assumptions were made for the assessment:

- The owner is the only privileged account in charge of coordinating each minting phase and setting up the current NFT contract.
- Each contract is deployed with the right parameters.
- The merkle tree is made off-chain and each whitelisted user is provided with a mean to get their index and proof.
- Whitelisted users can mint on the respective phase and in public mint, if there are tokens left.

4.2 Decentralization

The NFT contract has one permissioned method which is the minting function, which allows only executions made by the minter. As for the Investor Minter contract, an `owner` account has several management privileges: this `owner` can change the price of tokens (in ether) for both whitelisted and public phases, set a merkle root and the current minting phase, change the NFT contract, and transfer its ownership (in a single step).

4.3 Code quality & Testing

The code quality is up to modern standards and the codebase is easy to understand, including relevant comments and NatSpec. The project's tests are based on the Forge Framework. The tests provide several different scenarios that ease the process of developing proofs of concept for different potential attacks.

Coinspect identified that the test coverage for the `InvestorNFTMinterV1` contract can be increased, as it currently has a branch coverage of only 75%.

4.4 Threat Model

Coinspect built a threat model describing threats, attack scenarios and mitigations applicable to Le7el's contracts. This threat model was used as a guide when performing the review.

It is important to note that while it is an important component of a security review, the following threat model should not be considered exhaustive.

Threats and mitigations

Reentrancy attacks are one of the main threats to be considered, as these are common in NFT contracts. In particular, minting and transfer functions are usually targeted. Le7el mitigations include using reentrancy-guards on state-changing functions. It is important that this mitigation is kept in place in future upgrades to the contracts. Testing and static-analysis can also be implemented to improve the chances of finding any reentrancy-bugs.

Other important threats in NFTs contract are token-supply issues and problems arising from the use of `transferFrom()` or `mint()`. The project was found to use the safe-variant of these methods, but care must be taken to maintain this behavior in updates.

Another key threat are the interactions with arbitrary contracts, as the callees might be malicious. Coinspect found two issues related to this threat: `L7L-2` and `L7L-3`. It is important that care is taken to limit the amount of actions malicious contracts can take.

Broken access-controls were also taken into account. In particular, the `MINTER` address must be correctly set to ensure that only L7 investor minter is allowed to mint directly from the NFT contract. The same goes for the merkle root and the white-list that protects the minting mechanism: the root posted and the account's position must be the correct ones.

To make sure no mistakes occur here, dry-runs in Testnet or local networks can be done. These can be automated to make sure off-chain scripts that perform these actions work correctly.

Coinspect also considered common smart contracts threats such as arithmetic issues. Le7el contracts in-scope use modern Solidity versions which disallow overflows and operations were checked for correctness. Ensuring this remains in the future is important: continuous expansion of the test suite with border cases and fuzz-testing can aid uncovering these types of bugs quickly.

It is important that the contracts keep holding invariants related to many operations (for example, different phases allow different actions). To make sure that these invariants are always held, it is recommended to implement invariant testing.

Some threats were out of scope of this review, however, Coinspect recommends mitigations be put in place for this so as to ensure the continuous safety of the contract while it operates. One important aspect of the security of the project is the protection of the privileged `owner` address. Coinspect recommends this address to be managed through cold-storage or via a multisignature.

The Le7el whitepaper also mentions that a governance is implemented. Governances are a complex and powerful component of an off-chain system. Coinspect recommends that good practices such as timelocks and hash-code-equality between proposal and code-to-be-executed are implemented. Uniswap's Governance Seatbel can be used to lower the chance of malicious proposals being approved.

5. Detailed Findings

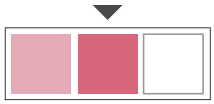
L7L-1

Whitelisted investors for new NFTs are not able to mint any token

Status
Solved

Resolution
Fixed

Risk
Medium



Impact
Low

Likelihood
High

Location

`InvestorNFTMinterV1.sol`

Description

When all whitelisted users mint an NFT, the `claimedBitMap` is filled with each user's `ID` depending on the `phase`. This bitmap is *NFT-agnostic*, meaning that for any NFT contract address, it will remain the same.

In other words, a first set of whitelisted users that mint an `NFTV1` populate the bitmap with their whitelist indexes (positions in the allow list), which is later reused for a new set of users minting an `NFTV2`.

```
    function _setClaimed(uint256 _index, uint256 _phase) private {
        uint256 _claimedWordIndex = _index / 256;
        uint256 _claimedBitIndex = _index % 256;
        claimedBitMap[_phase][_claimedWordIndex] =
claimedBitMap[_phase][_claimedWordIndex] | (1 << _claimedBitIndex);
    }
```

Consider a first set of whitelisted users with the following indexes:

1. `userA`
2. `userB`

They mint in the phase `0` an `NFTV1`, and the bitmap writes the `claimedBitMap` with the indexes of `0` and `1`.

Then, the `NFTV1` is changed for a `NFTV2`, and a new whitelist is created with the following users:

1. `userC`
2. `userD`

Because the indexes of `userA == userC` and `userB == userD`, the new users will not be able to mint the `NFTV2` token using the whitelist.

The impact of this issue is considered low, because a new minter instance could be deployed in the future patching this issue.

Proof of Concept

The following test shows how a first set of users mint on the whitelist phase an NFT. Then, the NFT address is changed and a new merkle tree is created with a new set of users. The test reverts when the first user of the new set tries to mint with the `Whitelist already used.` error string.

Output

```
  Trying to mint the first NFT...
  Merkle Root
  0x5b094a080d6c986749f6133184aec270dc727e979d2db5ca7d68e5ecc5ce57e8

User Index: 0
```

```
  Mint successful

User Index: 1
  Mint successful

NFT Contract changed.
  Trying to mint the new NFT...
  Merkle Root
  0xf72df6eb31c211e5a5abb66fb018754d4fda8bc8396b72e21938f332a6794eb7

User Index: 0

Failing tests:
Encountered 1 failing test in test/InvestorNFT.t.sol:InvestorNFTTest
[FAIL. Reason: Whitelist already used.] testNewNFTHasWLBlocked() (gas:
1320850)
```

Test

```
    function testNewNFTHasWLBlocked() public {
        // First, three users mint an NFT0 (first nft address)
        bytes32 _root = _merkle.getRoot(_merkleTreeLeafs);
        bytes32[][] memory proofs = new bytes32[][](2);
        for (uint256 i = 0; i < proofs.length; i++) {
            proofs[i] = _merkle.getProof(_merkleTreeLeafs, i);
        }
        address[] memory users = new address[](2);
        users[0] = _user1;
        users[1] = _user2;
        vm.prank(_admin);
        _investorNftMinter.ownerSetPhase(0, _root);
        console.log("Trying to mint the first NFT...");
        console.log("Merkle Root");
        console.logBytes32(_root);
        for (uint256 i = 0; i < users.length; i++) {
            console.log("\nUser Index: %s", i);
            vm.startPrank(users[i]);
            vm.deal(users[i], 0.1 ether);
            _investorNftMinter.mintTo{value: 0.049 ether}(users[i], i,
proofs[i]);
            vm.expectRevert("Whitelist already used.");
            _investorNftMinter.mintTo{value: 0.049 ether}(users[i], i,
proofs[i]);
            assertTrue(_investorNftMinter.isClaimed(i), "User not
claimed");
            vm.stopPrank();
            console.log("Mint successful");
        }
        // Then, a new NFT is created and the Minter replaces its
address
        InvestorNFTV1 _newInvestorNft = new
InvestorNFTV1(address(_investorNftMinter));
        vm.prank(_admin);

_investorNftMinter.ownerSetNFTContract(address(_newInvestorNft));
        console.log("\nNFT Contract changed.");
        // A new whitelist is created for this new NFT, with different
```

```
users
        delete _merkleTreeLeafs;
        address[] memory newUsers = new address[](2);
        bytes32[][] memory newProofs = new bytes32[][](2);
        newUsers[0] = address(1234);
        newUsers[1] = address(12345);
        _merkleTreeLeafs.push(keccak256(abi.encodePacked(uint256(0),
newUsers[0], uint256(1))));
        _merkleTreeLeafs.push(keccak256(abi.encodePacked(uint256(1),
newUsers[1], uint256(1))));
        newProofs[0] = _merkle.getProof(_merkleTreeLeafs, 0);
        newProofs[1] = _merkle.getProof(_merkleTreeLeafs, 1);
        bytes32 _newRoot = _merkle.getRoot(_merkleTreeLeafs);
        console.log("Trying to mint the new NFT...");
        console.log("Merkle Root");
        console.logBytes32(_newRoot);
        for (uint256 i = 0; i < newUsers.length; i++) {
            console.log("\nUser Index: %s", i);
            vm.startPrank(newUsers[i]);
            vm.deal(newUsers[i], 0.1 ether);
            // The following line will revert
            _investorNftMinter.mintTo{value: 0.049 ether}(newUsers[i],
i, newProofs[i]);
            vm.stopPrank();
            console.log("Mint successful");
        }
    }
```

Recommendation

Have separate `claimedBitMap` for each NFT address. Also, consider enforcing a merkle root update when replacing the NFT contract address to ensure that no previously whitelisted user is able to mint again when setting a new NFT contract.

Status

Fixed on commit `1f44804687a57c59b0dfbdda14133ba8fc148588`.

Since only one NFT contract address can be set, this issue is no longer feasible to happen.

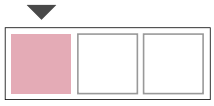
L7L-2

Malicious beneficiary can prevent anyone from minting

Status
Solved

Resolution
Fixed

Risk
Low



Impact
Medium

Likelihood
Low

Location

`InvestorNFTMinterV1.sol`

Description

A malicious beneficiary can harm minters by return bombing the minter preventing the call to conclude and as a consequence, halt the minting process.

The minting proceeds are sent to a `BENEFICIARY` account which is initialized at the contract's deployment, as an immutable variable.

This account can hold a malicious or an upgradable contract (capable of turning malicious in the future), that wastes minter's gas triggering a revert, disrupting the minting mechanism.

```
function _mintTo(address _account, uint256 _price, uint256 _index)
internal returns (uint256) {
```

```
        // Max id is checked on NFT contract level
        uint256 _i = currentIndex;
        currentIndex++;
        if (_price > 0) {
            require(msg.value >= _price, "not enough ETH to mint");
            (bool success,) = BENEFICIARY.call{value: msg.value}("");
            require(success, "ETH transfer failed.");
        }
        uint256 _id = _getId(_i);
        nftContract.mintTo(_account, _id);
        emit NewMint(_account, _id, _price, _index);
        return _id;
    }
```

When performing the low-level call, an unbound amount of gas is forwarded to the beneficiary, allowing the described scenario. It is considered that the beneficiary account will be trusted, thus the low likelihood of this scenario.

Proof of Concept

The following test shows an evil beneficiary that return bombs a minter by wasting all their gas. The mint call is reverted and the user is not capable to mint a token.

Output

```
Test result: FAILED. 0 passed; 1 failed; 0 skipped; finished in 1.55ms
Ran 1 test suites: 0 tests passed, 1 failed, 0 skipped (1 total tests)

Failing tests:
Encountered 1 failing test in test/InvestorNFT.t.sol:InvestorNFTTest
[FAIL. Reason: ETH transfer failed.] testReturnBombWhenMinting() (gas:
89373934605168165 60)
```

Test

```
    function testReturnBombWhenMinting() public {
        InvestorNFTMinterV1 _newInvestorNftMinter;
        InvestorNFTV1 _newInvestorNft;
        MaliciousBeneficiary _beneficiary;
        // Setup smart contracts
        vm.startPrank(_admin);
        _beneficiary = new MaliciousBeneficiary();
        _newInvestorNftMinter = new InvestorNFTMinterV1(_admin,
payable(_beneficiary), bytes32(0));
        _newInvestorNft = new
InvestorNFTV1(address(_newInvestorNftMinter));
```

```
_newInvestorNftMinter.ownerSetNFTContract(address(_newInvestorNft));
        _defaultWhitelist();
        vm.stopPrank();
        // Setup as public mint
        vm.startPrank(_admin);
        _newInvestorNftMinter.ownerSetPhase(2, bytes32(0));
        vm.stopPrank();
        // The Beneficiary return bombs a minter
        vm.startPrank(_user);
        vm.deal(_user, 0.1 ether);
        _newInvestorNftMinter.mintTo{value: 0.0649 ether}(_user);
        vm.stopPrank();
    }
```

With the following implementation for the `MaliciousBeneficiary` contract:

```
contract MaliciousBeneficiary {
    receive() external payable {
        uint256 rsize = _returnBombSender(gasleft() / 2 * 512);
        assembly {
            return(0x0, mul(rsize, 0x20))
        }
    }

function _returnBombSender(uint256 x) private pure returns (uint256 y)
{
        // Compute the sqrt(x)
        uint256 z = (x + 1) / 2;
        y = x;
        while (z < y) {
            y = z;
            z = (x / z + z) / 2;
        }
    }
}
```

Recommendation

Restrict the amount of return data that should be copied into memory. Nomad has an implementation for this, ExcessivelySafeCall. Alternatively, document the trust assumptions for the Beneficiary so users are aware of this risk.

Status

Fixed on `1f44804687a57c59b0dfbdda14133ba8fc148588`.

Documentation was added stating that this account will be a trusted multisig.

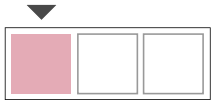
L7L-3

Users will waste gas when trying to mint after the investor deployment

Status
Solved

Resolution
Fixed

Risk
Low



Impact
Low

Likelihood
Low

Location

`InvestorNFTMinterV1.sol`

Description

When deploying the `InvestorNFTMinterV1` contract, a root and a beneficiary are provided. However, no NFT contract address is submitted. As the initial phase is set as `0` by default, if a valid root is committed whitelisted users trying to mint will waste gas. This happens because the `mintTo` call will revert as there is still no valid NFT contract.

```
    constructor(address _owner, address payable _beneficiary, bytes32
_merkleRoot) Owned(_owner) {
        BENEFICIARY = _beneficiary;
        merkleRoot = _merkleRoot;
        emit NewPhase(0, _merkleRoot);
        emit NewPrice(0.049 ether, 0.0649 ether);
    }
```

```
function _mintTo(address _account, uint256 _price, uint256 _index)
internal returns (uint256) {
        // Max id is checked on NFT contract level
        uint256 _i = currentIndex;
        currentIndex++;
        if (_price > 0) {
            require(msg.value >= _price, "not enough ETH to mint");
            (bool success,) = BENEFICIARY.call{value: msg.value}("");
            require(success, "ETH transfer failed.");
        }
        uint256 _id = _getId(_i);
        nftContract.mintTo(_account, _id); // <--------- this line will
revert when no contract was set
        emit NewMint(_account, _id, _price, _index);
        return _id;
    }
```

Recommendation

Create a new preparation phase where no one can mint. Alternatively, add an early return in the mint flow if no `nftContract` was set.

Status

Fixed on commit `1f44804687a57c59b0dfbdda14133ba8fc148588`.

A preparation phase where only the owner can mint was added.

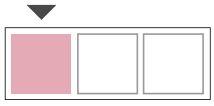
L7L-4

The owner can steal payments without minting any token

Status
Solved

Resolution
Fixed

Risk
Low



Impact
Medium

Likelihood
Low

Location

`InvestorNFTMinterV1.sol`

Description

The investor contract's owner can change the NFT address at anytime for a dummy NFT implementation that mints no token in return, effectively stealing the amount paid.

```
    function ownerSetNFTContract(address _nftContractAddress) external
onlyOwner {
        nftContract = IMintable(_nftContractAddress);
        emit NewNFTContract(_nftContractAddress);
    }
```

By changing the NFT address for another contract that respects the **IMintable** interface and implements an empty `mintTo` function, the minting process will

succeed but no tokens will be minted in exchange. Because users must provide the token's price upon minting which is transferred to the beneficiary, this process enables the owner to steal user's native tokens (considering that the owner controls the beneficiary).

Also, there are other key variables that can be changed at anytime with no restriction by the owner such as the merkle root, token price and minting phase.

Proof of Concept

The following test shows how the owner is able to steal the amount paid by a user by simply changing the NFT address for a Fake NFT. This process of changing the NFT address can be done at anytime, as there are no time restrictions for the owner to setup a new address.

Test

```
contract MaliciousNFT {
    function mintTo(address, uint256) external {}
}

    function testUseFakeNFTToStealUsersTokens() public {
        FakeNFT fakeNft = new FakeNFT();
        // Setup new fake nft address and public mint
        vm.startPrank(_admin);
        _investorNftMinter.ownerSetNFTContract(address(fakeNft));
        _investorNftMinter.ownerSetPhase(2, bytes32(0));
        vm.stopPrank();
        vm.startPrank(_user);
        vm.deal(_user, 0.1 ether);
        _investorNftMinter.mintTo{value: 0.0649 ether}(_user);
        assertTrue(_user.balance == 0.0351 ether);
        assertTrue(_investorNftMinter.BENEFICIARY().balance == 0.0649
ether);
        vm.stopPrank();
    }
```

Recommendation

Add a time-lock to the NFT address setter. Also, evaluate time-locking the other privileged setters for a more transparent contract management.

Status

Fixed on `1f44804687a57c59b0dfbdda14133ba8fc148588`.

The NFT implementation now can be set only once.

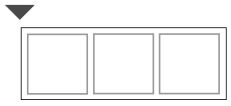
L7L-5

Users can unsuspectingly mint the wrong token

Status
Solved

Resolution
Fixed

Risk
None



Impact
Recommendation

Likelihood
–

Location

`InvestorNFTMinterV1.sol`

Description

Users can only check the current NFT address by filtering the `NewNFTContract` event. There is no simple way for users to check the current `nftContract` directly from the `InvestorNFTMinterV1` contract interface as it is private and there is no getter for its value.

Additionally, because the NFT address can change, there is a collision chance for the `NewMint` event for two different tokens:

```
event NewMint(address indexed owner, uint256 indexed nftId, uint256
price, uint256 merkleIndex);
```

Two identical events for an user minting different NFTs (contracts) with the same `id`, `price` and `merkleIndex` will be emitted.

Recommendation

Allow users to check the current `nftContract` with a getter or making it public. Add the `nftContract` address to the `NewMint` event.

Status

Fixed on `1f44804687a57c59b0dfbdda14133ba8fc148588`.

The `nftContract` variable was made public.

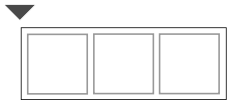
L7L-6

Users can precalculate the NFT ID as the semi-random generation has no randomness

Status
Solved

Resolution
Fixed

Risk
None



Impact
Recommendation

Likelihood
–

Location

`InvestorNFTMinterV1.sol`

Description

Users wanting a specific NFT `ID` can mint a token that matches the desired `ID` by waiting until the `_getId()` function returns the expected value.

Each NFT `ID` is retrieved from a list that has mixed values up to 2222 (`_ALL_IDS`). The spec. of the ID generation process specifies that it is retrieved using a **semi-random** process. However, this process has no randomness because it only extracts 16-bit values from the `_ALL_IDS` array.

The ID generation process relies on a sequentially increased index which is used as the position to extract a value from the `_ALL_IDS` array:

`_mintTo()`:

```
        uint256 _i = currentIndex;
        currentIndex++;

{...}

uint256 _id = _getId(_i);
```

Then, `_getId()`:

```
    /**
     * @dev Based on _i offset extract id of NFT.
     *
     * @param _i Index of token id.
     * @return semi-random NFT id.
     */
    function _getId(uint256 _i) private pure returns (uint16) {
        uint16 _id;
        bytes memory _ids = _ALL_IDS;
        // solhint-disable-next-line no-inline-assembly
        assembly {
            _id := and(mload(add(_ids, mul(_i, 2))), 0xFFFF)
        }
        return _id;
    }
```

Users can precalculate without effort the NFT `ID` because:

1. the `_ALL_IDS` array is populated with all numbers from 1 to 2222, scrambled
2. and the `currentIndex` increases after each mint

As long as the `_ALL_IDS` remains the same, for every NFT contract the order on which each ID is minted will be always the same but in no linear order.

This issue is considered informational as the Le7el team stated that there is no security risk attached to the `ID` value and utility is the same for all `IDs`.

Recommendation

Document this mechanism to get the `IDs` in the spec explaining the generation process.

For other implementations or applications where a more robust randomness structure is needed, use a **VRF (Verifiable Random Function)** oracle to generate random `IDs`.

Status

Fixed on `1f44804687a57c59b0dfbdda14133ba8fc148588`.

The ID generation mechanism is now documented.

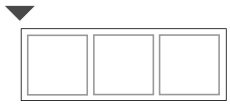
L7L-7

Minting phases can be replaced by an enum

Status
Solved

Resolution
Fixed

Risk
None



Impact
Recommendation

Likelihood
–

Location

`InvestorNFTMinterV1.sol`

Description

Minting phases can be expressed with an `enum` naming its members after each minting phase for better understanding.

Currently, the minting phases are expressed with an `uint256`:

```
    uint256 public phase = 0;
```

This could be replaced, for example for the following `enum`:

```
    enum MintingPhases {
        GTD,
        FCFS,
```

```
        PUBLIC
    }
```

Recommendation

Use an `enum` to determine each minting phase.

Status

Fixed on `1f44804687a57c59b0dfbdda14133ba8fc148588`.

Documentation explaining each phase was added to the contract.

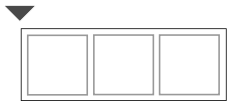
L7L-8

Ownership can be mistakenly lost

Status
Caution Advised

Resolution
Open

Risk
None



Impact
Recommendation

Likelihood
–

Location

`InvestorNFTMinterV1.sol`

Description

The Investor Minter contract is owned by a single account which has the privileges to:

1. Change the NFT address
2. Set a new price for both public and whitelisted phases
3. Set the current merkle root and minting phase
4. Transfer the ownership

The ownership transfer process is made on a single step, meaning that in the event of passing the wrong address control over the mentioned actions will be irreversibly lost:

```
    function transferOwnership(address newOwner) public virtual
onlyOwner {
```

```
        owner = newOwner;

        emit OwnershipTransferred(msg.sender, newOwner);
    }
```

Recommendation

Use a two-step ownership transfer mechanism.

Status

The Le7el team stated that ownership would have no use after initial 2222 NFTs are minted.

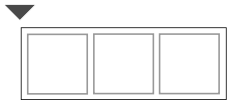
L7L-9

Different instances of the Investor NFT will have the same name and symbol

Status
Solved

Resolution
Deferred

Risk
None



Impact
Recommendation

Likelihood
–

Location

`InvestorNFTMinterV1.sol`

Description

Deploying different instances of the same Investor NFT, will result on tokens with the same metadata. This can make it difficult to quickly identify different instances of a Le7el token when looking at transactions using a blockchain explorer.

Blockchain explorers rely on the `name()` and `symbol()` functions of the contract's interface to display the token's name and symbol on their frontend. The Investor token contract name suggests that in the future a `V2` might be deployed (`InvestorNFTV1`). Currently the token is deployed with the following metadata:

```
    constructor(address _minter) ERC721("L7 Investors", "L7IN") {
        MINTER = _minter;
```

```
    }
```

Meaning that in the event of deploying a `V2` of this token, the naming convention might not be respected for future versions.

Recommendation

Use a more distinctive name and symbol for the `V1` of the token, in the event of expecting to deploy more versions in the future.

Status

The Le7el team stated that they don't plan to deploy new versions of this NFT in a future, and they will handle changes on metadata level.

6. Disclaimer

The information presented in this document is provided "as is" and without warranty. The present security audit does not cover any on-chain systems or frontends that communicate with the network, nor the general operational security of the organization that developed the code.