ooNodz
Smart Contracts Audit

Date	Author	Notes
2023-06-28	Antoine Detante	Initial version
2023-07-14	Antoine Detante	Addition of 2 new “warnings”
2023-09-26	Antoine Detante	Updated with fixed version

1. Introduction

1.1. Context

ooNodz provides infrastructure services for users to operate validation nodes on the Avalanche blockchain. More specifically, the company has developed a blockchain protocol enabling its customers to provision validation nodes on the network, for a fixed & prepaid period of time.
As a non-custodial hosting provider, the solution does not manage the customers' stake: only the infrastructure component (i.e. validation nodes) is operated by the platform, leaving customers in full control of their funds.

The protocol audited in this document refers to the Smart Contracts used by customers to register a validation node for a given period of time.

URL of the audited Smart Contracts:
https://gitlab.com/oonodz/contracts/-/tree/7b3fb1ef8ed1102193344543a12afc024bf60532/contracts
Git commit: `7b3fb1ef8ed1102193344543a12afc024bf60532`

1.2. Objectives

The objective of this audit is to analyze the source code of Smart Contracts in order to:

- Identify any potential security risks (code bugs, malicious use, well-known security vulnerabilities, ...)

- Assess the maintainability, readability and compliance with industry standards (in the perspective of the platform's future developments)

2. Audit Methodology

The methodology used to produce this audit was based on the following principles:

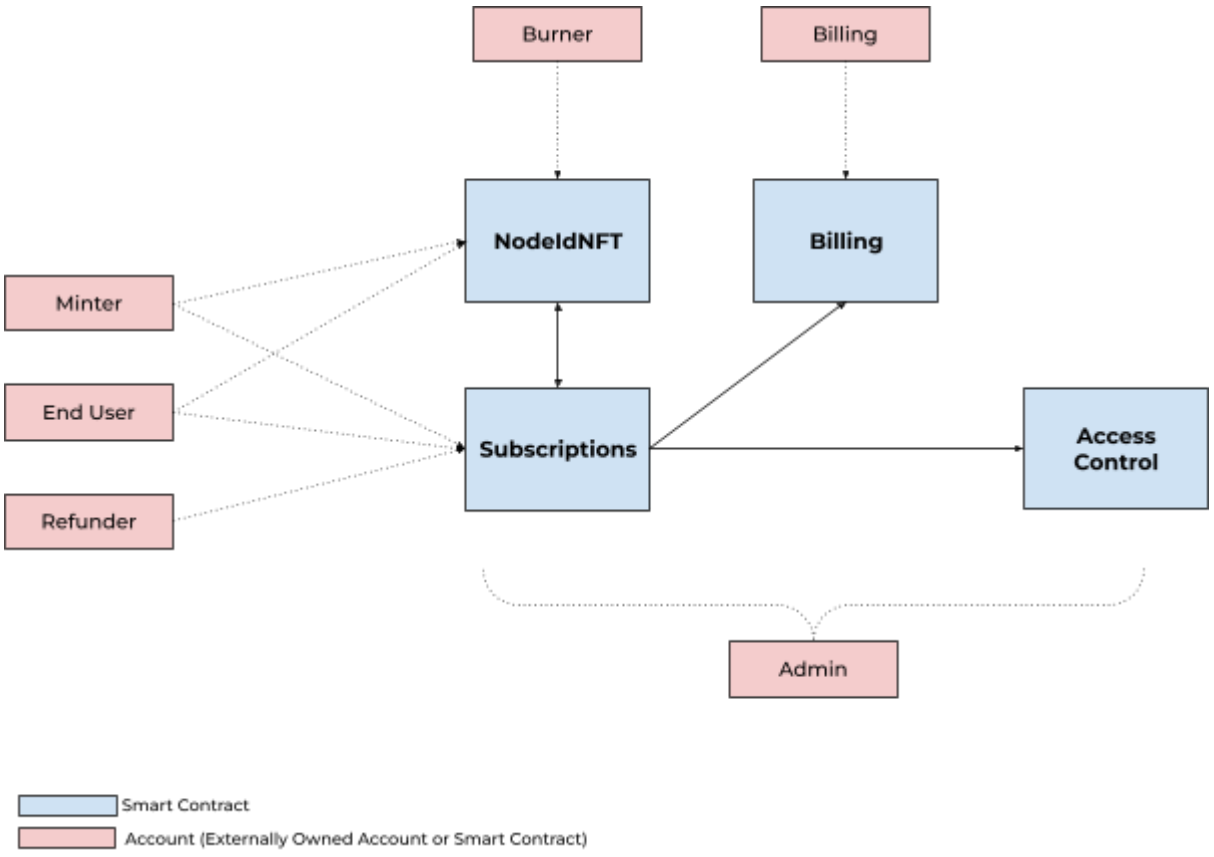
- **Code Review**: Perform a review of the Solidity code to identify any potential security vulnerabilities. More specifically:
    - Input Validation (check if the contract properly validates and sanitizes user inputs)

    - Access Control (ensure that the contract implements appropriate access control mechanisms to prevent unauthorized access)
    - Function and Variable Visibility
    - External Calls and Contracts Interaction (assess how the Smart Contracts interact with other contracts)
    - Error Handling
    - Security Best Practices (ensure the code follows Solidity best practices)
- **Tests**
    - Analysis of existing unit tests
    - Execution of unit tests
    - Manual execution of Smart Contracts on a local environment to analyze operation and interactions between contracts
- **Static Analysis**: use static analysis tools to detect potential vulnerabilities or suspicious patterns. These tools can help identify common security issues like uninitialized variables, dead code, or code that can be prone to vulnerabilities.
- **Manual Review and Verification** to validate the findings from the previous steps

3. Smart Contracts & Roles Overview

The protocol is structured around 4 Smart Contracts that are linked to each other:

- **AccessControl**, based on standard OpenZeppelin Access Control model, used to define and check roles and authorizations of accounts that are involved with the platform
- **NodeIdNFT**, based on OpenZeppelin implementation of standard ERC721 Token, used to represent a validator identifier (NodeId) and its current owner
- **Billing** used to store billing configuration (VAT rates, subscription prices, supported ERC20 tokens for payment, ...)
- **Subscriptions** is the protocol's main Smart Contract, with which end users interact directly. It allows an user to start, extend or cancel a service subscription.



In addition to End Users who subscribe to the service, 5 roles have been defined and are used to control access to Smart Contract functionalities:

- **Admin:** the admin's role is limited to Smart Contract configuration functions (link between contracts, term & condition versions). His main responsibility is to define the address of the external wallet that will receive all user payments.
- **Billing**: accounts with the Billing role can update billing config (VAT rate, tokens supported for payment, service price, ...)
- **Burner**: a special role that can (in addition to the current token's owner) destroy a given NodeIdNFT token
- **Minter**: accounts with the Minter role can create NodeIdNFT tokens, before these are distributed to End Users. They are also responsible for defining the number of validation slots available (i.e. limiting the number of new subscriptions).
- **Refunder**: "refunders" can finalize end user's refund requests (thus triggering the transfer of ERC20 tokens to be refunded to the user's account).

This separation of roles provides a clear division of responsibilities between the different aspects of the system.

4. Upgradability

These 4 Smart Contracts deployed for the ooNodz platform are **upgradable**.
The solidity code can therefore be updated after the initial deployment, via a “Proxy” mechanism that separates the Smart Contract implementation and the addresses used to interact with it.

In this way, the current version of the implementation is configured at the Proxy level, and a special account (*admin*) can be used to change the address of the current implementation of each of the 4 Smart Contracts.

This solution has multiple advantages (easy maintenance of Smart Contracts, in particular quick and transparent fix if a bug is detected in the code of a Smart Contract), but also has an impact in terms of transparency: the rules defined in Smart Contracts can be modified by the administrator.

In this case, Smart Contracts do not store user funds (user stakes are managed directly by the Avalanche protocol, outside the ooNodz platform, via the Avalanche Wallet). So **these upgrades of Smart Contracts have no impact on user funds**, and do not introduce any particular risk for the user.

The implementation chosen here is based on Openzeppelin's Upgradable library. Smart Contract administration operations (contracts’ upgrade, change of proxy address, ...) are handled by the OpenZeppelin Defender platform and the Safe wallet.
⇒ Gnosis Safe at address `0x1fC01374A2A2375cFa93478c2bE459F98A6355d8`.

In this way, Smart Contract upgrade operations are operated within the current ooNodz governance framework: the “admin” account is protected by the Vault, which requires a multisig from project co-founders before triggering a Smart Contract update.

This limits the risk of a unilateral takeover by one of the parties, and is also scalable according to the needs of the project: **a decentralized governance can then be introduced**, with for example a stakeholder voting mechanism to authorize an upgrade of the Smart Contracts.

5. Findings & Recommendations

5.1. Severity Breakdown

Level	Description	Recommended action
Critical	Bugs that may result in the theft of assets, the locking of funds, or any other scenarios where funds could be transferred to unauthorized parties.	Immediately fix the issue
Major	Bugs that have the potential to cause a contract failure, requiring manual intervention to recover or modify the contract state	Fix as soon as possible
Warning	Bugs that can disrupt the intended logic of the contract or make it vulnerable to DoS attacks	Take into consideration and implement fix in certain period
Comment	Other issues and recommendations reported	Take into consideration.

After receiving feedback from the development team regarding the list of findings listed in this Audit Report, the following statuses have been assigned to each finding:

Status	Description
Fixed	Recommended fixes have been made to the code and no longer affect its security
Acknowledged	The team acknowledges this finding. Plans are in place to address the recommendations associated with this finding in the future. It is determined that this finding does not impact the overall safety of the Smart Contract
No issue	Finding does not affect the overall safety of the Smart Contract

5.2. Findings

5.2.1. **Critical**

No Critical point found.

5.2.2. **Major**

No Major point found

5.2.3. **Warning**

WARNING-1	Potential Re-Entrancy vulnerability in completeRefund
Smart Contract	Subscriptions.sol
Status	**Fixed** `(68fd38a712614c9962b8104c8071f56daa74ac87)`

Description

In the `completeRefund` function, the call to `transferFrom` to transfer the refund to the user is done *before* updating the state:

```
// Transfer tokens to customer
token.transferFrom(_msgSender(), _wallet, _amount);

// Change refund status
refunds[_wallet][_tokenId].pending = false;

emit RefundCompleted(_wallet, _msgSender(), _tokenId, refund.currencySymbol,
refund.amount);
```

If the target ERC20 Smart Contract allows calls to be made to an external function, then it is potentially possible to perform a re-entrancy attack on this function.
In practice, the risk of this flaw is linked to non-standard ERC20 Smart Contracts being authorized on the Billing contract. This is not currently the case, so this function is not vulnerable to a re-entrancy attack with the USDT and USDC tokens currently supported.

See here for more information about Re-Entrancy.

WARNING-1	Potential Re-Entrancy vulnerability in completeRefund

Recommendation

Move state update

```
// Change refund status
refunds[_wallet][_tokenId].pending = false;
```

before the call to `token.transferFrom`

WARNING-2	Pending Refund is discarded when user starts a new subscription
Smart Contract	Subscriptions.sol
Status	**Fixed** `(68fd38a712614c9962b8104c8071f56daa74ac87)`

Description

In the `hasPendingRefund` function, the code checks that the user has no active subscription, otherwise it is assumed that there is no pending refund:

```
function hasPendingRefund(address _wallet, uint256 _tokenId) public view returns
(bool) {
  // Cannot have a pending refund if the subscription is still active
  if (hasActiveSubscription(_wallet, _tokenId) == true) return false;
  // ...
}
```

However, it is possible to create a new subscription with a pending refund.
In this case, if the user creates a new subscription before the previous refund is completed, the previous refund is discarded.

Note: this bug impacts functionality for the user, not Smart Contract security.

Recommendation

Remove the test about active subscription in `hasPendingRefund` function.

WARNING-3	Yearly subscription reload conditions not checked
Smart Contract	Subscriptions.sol
Status	**Fixed** (68fd38a712614c9962b8104c8071f56daa74ac87)

Description

When reloading a yearly subscription, the contract must ensure that the renewal occurred in the last month of the current subscription, and for a maximum duration of 1 year.

Currently these rules are controlled using this test:

```
uint256 _startTs = currentSubscription.endTs;
uint256 _endTs = currentSubscription.endTs;

// ...
if (currentSubscription.period == Billing.SubscriptionPeriod.Year) {
  _endTs = _startTs + (365 days) * uint256(_addedPeriods);
  require(
    (_endTs - _startTs) < 396 days,
    "a yearly subscription can only be reloaded the last month and only for one more year"
  );
}
```

If the user chooses to reload for 1 year, then he can reload even if he is not in the last month of the current subscription. As a result, he can extend his subscription by several years, which is not expected.

Recommendation

In the `require` test, check the duration against the end date of the reloaded subscription **and the current date** (`block.timestamp`) to ensure that the new end date does not exceed 1 year and 1 month (= 396 days).

```
  require(
    (_endTs - block.timestamp) < 396 days,
    "a yearly subscription can only be reloaded the last month and only for one more year"
  );
```

5.2.4. **Comment**

COMMENT-1	Potential Re-Entrancy vulnerability in newSubscription
Smart Contract	Subscriptions.sol
Status	**Fixed** (68fd38a712614c9962b8104c8071f56daa74ac87)

Description

In the `newSubscription` function, the call to `transferFrom` to transfer the refund to the user is done *before* updating the state:

```
// Transfer amount to destinationWallet
token.transferFrom(_msgSender(), destinationWallet, totalTokenAmount); // ICI

// Store subscription
uint256 _priceRate = billing.priceRates(uint8(currentSubscription.period));
uint256 _vatRate = billing.vatRates(customers[_msgSender()].countryOfResidence);

subscriptions[_msgSender()][_tokenId].endTs = _endTs;
subscriptions[_msgSender()][_tokenId].priceRate = _priceRate;
subscriptions[_msgSender()][_tokenId].vatRate = _vatRate;

nodeidFreeAfter[_tokenId] = _endTs;
```

But the target address (`destinationWallet`) is supposed to be under team control so the impact is non-existent.

Recommendation

Move call to `transferFrom` after state updates

```
// Store subscription
uint256 _priceRate = billing.priceRates(uint8(currentSubscription.period));
uint256 _vatRate = billing.vatRates(customers[_msgSender()].countryOfResidence);

subscriptions[_msgSender()][_tokenId].endTs = _endTs;
subscriptions[_msgSender()][_tokenId].priceRate = _priceRate;
subscriptions[_msgSender()][_tokenId].vatRate = _vatRate;

nodeidFreeAfter[_tokenId] = _endTs;

// Transfer amount to destinationWallet
token.transferFrom(_msgSender(), destinationWallet, totalTokenAmount);
```

COMMENT-2	Returned value of transferFrom is not used
Smart Contract	Subscriptions.sol - newSubscription Subscriptions.sol - reloadSubscription Subscriptions.sol - completeRefund
Status	**Fixed** `(68fd38a712614c9962b8104c8071f56daa74ac87)`

Description

The ERC20.transferFrom function returns a boolean indicating if the operation succeeded.
This value is not checked by the Smart Contract.
But mainnet implementations of ERC20 (USDC and USDT) revert if the transfer fails, so there's no problem at the moment.

Recommendation

Check that the call to `transferFrom` returns `true`

```
// --- FOR EXAMPLE ---
// Transfer amount to destinationWallet
require(
  token.transferFrom(_msgSender(), destinationWallet, totalTokenAmount),
  "error occurred during transferFrom"
);
```

6. Conclusion

In conclusion, **no significant security risks have been identified** in the Smart Contracts audited.

It is recommended to apply the few recommendations listed in this document. They do not imply any current risk, but they will make the contracts more resilient to future changes in the external contracts on which they depend.

The use of standard libraries (OpenZeppelin for Access Control & modeling of the ERC721 token), common tools in the ecosystem (Hardhat & Solhint) and unit tests **guarantee a high level of security & quality in the code** and readability of Smart Contracts for those involved in the source code.

Best practices are applied, **the code is clear, easy to understand and comfortable to read**.

Appendix 1 : Additional Static Analysis Results

File	Topic	Note
NodeIdNFT.sol	NodeIdNFT.refund(address) (contracts/NodeIdNFT.sol) sends eth to arbitrary user Dangerous calls: - user.transfer(__refund) (contracts/NodeIdNFT.sol)	The refund destination address (user) is checked in the same function to verify that there is a pending refund for this user.
Subscriptions.sol	computeRefundPeriodsOf(uint256,address) A control flow decision is made based on The block.timestamp environment variable. The block.timestamp environment variable is used to determine a control flow decision. Note that the values of variables like coinbase, gaslimit, block number and timestamp are predictable and can be manipulated by a malicious miner. Also keep in mind that attackers know hashes of earlier blocks. Don't use any of those environment variables as sources of randomness and be aware that use of these variables introduces a certain level of trust into miners.	The timestamp can be manipulated by miners, but the time lag can only be a few minutes, otherwise the block is no longer valid and is rejected. In the use case presented here, this manipulation of a few seconds is not a problem.