Smart Contract Security Audit

Novem

2020-11-30

Content

1. Introduction

Novem is set to transform the precious metal industry using high ethical standards, modern communications and an advanced blockchain technology. Novem offers physical products sold in their stores, as well as their two digital crypto currency tokens.

The physical products comprise LBMA certified gold bars of all dimensions and gold products like jewelry and gold gift cards. These products can be bought with traditional fiat money, with gold and, when paid for with their utility token, a noticeable discount applies.

As requested by Novem and as part of the vulnerability review and management process, Red4Sec has been asked to perform a security code audit and a cryptographic assessment in order to evaluate the security of the Novem Smart Contract source code on Ethereum Blockchain.

2. Scope

The scope of this evaluation includes:

Ethereum Novem Smart Contract

- NNNToken.sol
- SHA256:
  `9ab58a99da9cfcdfac77492f64a42330c9e82ec57eb80dacba1a066bdbc150de`
- https://github.com/overqint/erc20_nnn/tree/f36fdfc22d0246840d33a805960f052194595554

3. Conclusions

To this date, 30th of November 2020, the general conclusion resulting from the conducted audit, is that the **Novem Smart Contract is secure,** but the project needs to improve some points which are detailed below. Additionally, Red4Sec has found a few potential improvements, these do not pose any risk by themselves and we have classified such issues as informative only, but they will help Novem to continue to improve the security and quality of its developments.

- The **quality of the Unit Test** could be improved. The development cycle needs to be reviewed since we have found that some unit tests are not performing satisfactorily.
- The logic of Transfer fee in the Novem smart contract does not contemplate all the possible scenarios and **should be reviewed thoroughly**.
- A **few low impact issues** were detected and classified only as informative, but they will continue to help Novem improve the security and quality of its developments.

4. Issues & Recommendations

Contracts Management Risks

The logic design of the Novem contracts imply a few minor risks that should be reviewed and considered for their improvement and safety.

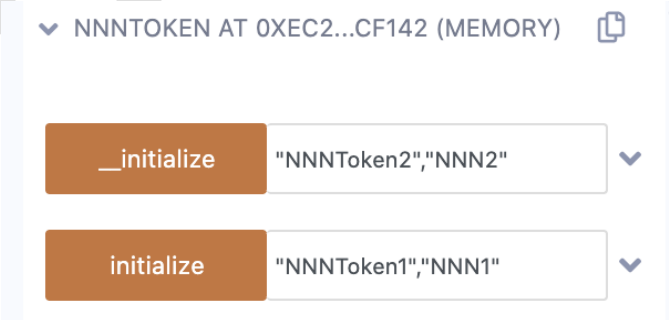
Possible wrong initialization

It has been verified that two initialization methods are exposed during the compilation of the contract, which are: *__initialize* and *initialize*. Eliminating the *initialize* method is convenient since there is a possibility that, due to human error, the contract can be initialized by said method, which would avoid calling the initialization code corresponding to *__EnhancedMinterPauser_init_unchained.*

```
function __EnhancedMinterPauser_init_unchained() internal initializer {
    _setupRole(FEE_EXCLUDED_ROLE, _msgSender());
    setMintingFeeAddress(0xFEff5513B45A48D0De4f5e277eD22973a9389e0B);
    setTransferFeeDivisor(2000);
}
```

We should always remember that the code of the initializers is a code generated within the contract's bytecode, alternatively the one in the constructor is a code that is not stored in it after the execution of the deployed transaction.

So, in the inheritance's case, it is convenient to make the initializers, that we don't intend to make public, as internal; because, as it can be observed, multiple overloads of the same method will be generated and this may induce unwanted errors, if called in a different order than expected.



In order to avoid exposure to unwanted methods, it is recommended to declare the unwanted one as internal.

Possible front-running in initialize methods

The initialize method used to establish the contract's parameters after the deploy, may be invoked by any user. Anyone can front-run the ***initialize()*** function right after the deploy and set themselves as owner of the new contract. Therefore, it is convenient to limit the initialize methods to the deployer or to put special attention during the contract's deployment to ensure that nobody calls ***initialize*** before us.

Wrong logic around Transfer fee

The divisions made by the Safe Math library prevent these to be executed over *0* since the result would be infinite and this value cannot be managed by the Ethereum virtual machine.

```
return div(a, b, "SafeMath: division by zero");
```

When making transfers there is the possibility of defining a *tokenTransferFeeDivisor* fee, the logic related to the sending process of this fee does not control that it could be *0*, which would trigger an operation revert during the user's transfers.

```
// calculate transfer fee and send to predefined wallet
function _calculateAmountSubTransferFee(uint256 amount)
    private
    returns (uint256)
{
    //using SafeMath for uint256 transferFeeAmount = amount.div(tokenTransferFeeDivisor);
    super.transfer(feeAddress, amount.div(tokenTransferFeeDivisor));
    return amount.sub(amount.div(tokenTransferFeeDivisor));
}
```

If at any time the owner of the contract decides to set this value to 0, or calls the wrong initializer (*initialize*), it will cause a denial of service as can be seen in the following image:

```
transact to NNNToken.transfer errored: VM error: revert. revert The transaction has
been reverted to the initial state. Reason provided by the contract: "SafeMath:
division by zero". Debug the transaction to get more information.
```

Source reference:

- https://github.com/overgint/erc20_nnn/blob/f36fdfc22d0246840d33a805960f052194595554/contracts/EnhancedMinterPauser.sol#L103

Fixed decimals

A highly recommended practice is the use of constants since it results less expensive when the code is executed. However, in this scenario, the constant refers to a value that must be defined when building the contract. Therefore, it is recommended to make a call to the *decimals* function to obtain the actual value, this will prevent to assume that it is 18 decimals.

```
// minting process does not involve fees (by design)
function mintWithoutDecimals(address recipient, uint256 amount) public {
    require(
        hasRole(DEFAULT_ADMIN_ROLE, _msgSender()),
        "Caller must have admin role to mint"
    );

    return super._mint(recipient, amount * 1000000000000000000);
}
```

Source reference:

- https://github.com/overgint/erc20_nnn/blob/f36fdfc22d0246840d33a805960f052194595554/contracts/EnhancedMinterPauser.sol#L49

Provide License for Third-Party Code

The audited contract inherits the functionalities of OpenZeppelin contracts however, it has been included in the code by copying them rather than by package manager. This is not recommended by OpenZeppelin, though not necessarily incorrect.

By using the original sources, in case the project resolves any vulnerability or bug in the code, we would obtain this update automatically, avoiding inheriting known vulnerabilities. Additionally, these OpenZeppelin contracts are under the MIT license, which requires its license/copyright to be included within the code.

For this reason, we highly recommend including the reference or copyright in the audited project.

Recommendations:

- Include third-party codes by package manager.
- Include any references/copyright to OpenZeppelin code in the Novem project, since it is under MIT license.

Outdated Compiler Version

Solc frequently launches new versions of the compiler. Using an outdated version of the compiler can be problematic, especially if there are errors that have been made public or known vulnerabilities that affect such version.

The audited contracts and its compilation script use *0.7.0* solidity version:

```
1  pragma solidity 0.7.0;
2
3  import "./EnhancedMinterPauser.sol";
```

Solidity branch *0.7* has important bug fixes in the array processing, so it is recommended to use the most up to date version of the pragma.

Source references:

- https://github.com/overqint/erc20_nnn/blob/f36fdfc22d0246840d33a805960f052194595554/contracts/NNNToken.sol#L1
- https://github.com/ethereum/solidity/blob/develop/Changelog.md#074-2020-10-19

Invest in Security, invest in your future