Smart Contract Security Audit

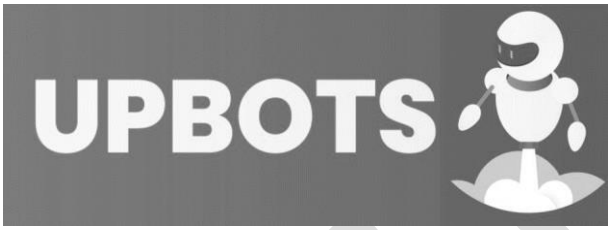
UpBots

2020-11-04

Content

1. Introduction

Upbots is an all-in-one platform that brings together the best crypto trading tools and strategies that are generally stand-alone services. It provides a 360° trading experience where users simply choose what best suits their profile.



At **UpBots** their mission is to create an all-in-one platform that democratizes the financial revolution that Bitcoin started, and which decentralized finance is now expanding upon.

With class leading tools designed to empower their success whether newbie or advanced, whether trading crypto on centralized or decentralized exchanges, or even investing in DeFi solutions.

As requested by **UpBots** and as part of the vulnerability review and management process, **Red4Sec** has been asked to perform a security code audit in order **to evaluate the security of the UpBots Smart Contract source code.**

2. Scope

UpBots Smart Contract:

- OwnedUpgradeabilityProxy.sol
  - SHA256:
    cacdfffdf62a8a5881e3f987b1200b2535ac2d710c30305bfcff5fa0f74fbbf4
- TimelockExtendable.sol
  - SHA256:
    994d7eaad48db237b89e81c722e793fad2a6b5556d69d01f430effaf7d7f0f31
- TokenRedeem.sol
  - SHA256:
    aac529996524761f40c132dc8e8343c864dd9b8961378e0ee070408be51c4899
- UbxToken.sol
  - SHA256:
    c2d58b0f2c5f5b2a71cb6eb57fc6b6812583acee332642ddb9ddb2759e164aa5

3. Conclusions

To this date, 4th of November 2020, the general conclusion resulting from the conducted audit, is that **UpBots Smart Contracts do not present any known vulnerabilities** that could compromise the security of the users and their information, although Red4Sec has found a few potential improvements, these do not pose any risk by themselves.

The general conclusions of the performed audit are:

- A few **low impact issues** were detected and classified only as informative, but they will continue to help UpBots improve the security and quality of its developments.
- The analyzed smart contracts **from the UpBots project comply with good development practices**; they have good organization, comprehensive controls, a good unit test battery, upgrade functions and a correct modularization of the project.
- While the contracts have administrative functions that allow complete control of the project, it also **includes a logic that allows** you to transfer these permissions to Time Lock, Multisign or even to give them up, ensuring **the complete decentralization of the project.**

4. Recommendations

Contracts Management Risks

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

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.

Possible loss of token

Even though this logic is intentional, it is necessary to mention that the UpBots token allows to pause its functionality (*transfer, burn…*) and afterwards to revoke or give up the owner's role and the *pausable* role, which would leave the token permanently and irrevocably useless.

Wrong Visibility

In order to simplify the contract for the users, it is recommended to turn the following variables to public.

```
// ERC20 basic token contract being held
IERC20 private _token;

// beneficiary of tokens after they are released
address private _beneficiary;

// timestamp when token release is enabled
uint256 private _releaseTime;
```

When initializing variables as public the following methods will no longer be necessary, and can be deleted:

```
function token() public view returns (IERC20) {
    return _token;
}

/**
 * @return the beneficiary of the tokens.
 */
function beneficiary() public view returns (address) {
    return _beneficiary;
}

/**
 * @return the time when the tokens are released.
 */
function releaseTime() public view returns (uint256) {
    return _releaseTime;
}
```

References:

- https://github.com/upbots/smart-contracts/blob/1449da13ae44ad399f7293c0bd1b941f345a9c57/contracts/TimelockExtendable.sol#L47-L66

Code Style

It has been possible to verify that, despite good quality code, there are a few modifications that can help make the code more understandable and easier to analyze.

In order to unify the pattern to make the comprehension and use of the contract simpler for the users, we recommend that the contracts *CanReclaimEther.sol* and *CanReclaimToken.sol* use the same logic.

As you can see below, in the *CanReclaimEther.sol* contract the ***reclaimEther()*** method contains the *onlyOwner* modifier, besides, when using the *transfer* method, it is directly done with the calling address (***address(this)***):

```
function reclaimEther() external onlyOwner {
    msg.sender.transfer(address(this).balance);
}
```

However, in *CanReclaimToken.sol* you can see how, as stated in the previous example, the ***reclaimToken()*** method implements the *onlyOwner* modifier. Nonetheless when the transfer is made, the **owner()** method is called when ***address(this)*** could be used because the *onlyOwner* modifier will force only one owner to invoke that method.

This is not a vulnerability by itself, but it helps to improve the code and reduce the rise of new issues.

```
function reclaimToken(IERC20 token) external onlyOwner {
    uint256 balance = token.balanceOf(address(this));
    token.safeTransfer(owner(), balance);
}
```

References:

- https://github.com/upbots/smart-contracts/blob/1449da13ae44ad399f7293c0bd1b941f345a9c57/contracts/utils/reclaim/CanReclaimEther.sol#L11
- https://github.com/upbots/smart-contracts/blob/1449da13ae44ad399f7293c0bd1b941f345a9c57/contracts/utils/reclaim/CanReclaimToken.sol#L14

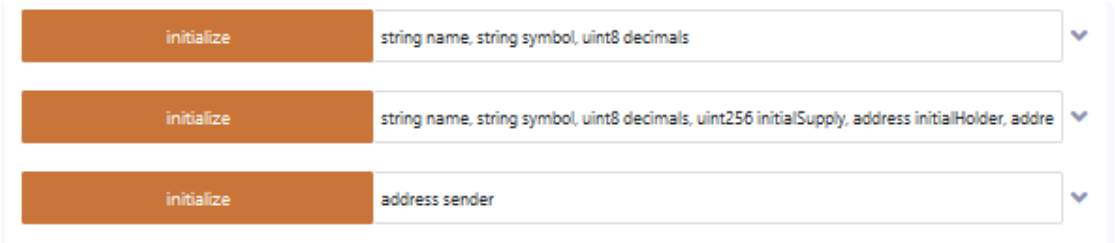
Multiple initializers exposed

The contracts have been designed to be resilient to updates through the use of proxies, therefore the use of ***initialize*** methods have been decided over constructors[1], this logic results in a higher GAS expense, but it is necessary under certain circumstances.

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

[1] https://docs.openzeppelin.com/upgrades-plugins/1.x/writing-upgradeable

that is not stored in it after the execution of the deploys 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 *Ownable, Pausable* and *ERC20* initializers as internal.

Gas Optimization

Software optimization is the process of modifying a software system to make an aspect of it work more efficiently or use less resources. This premise must be applied to smart contracts as well, so that they execute faster or in order to save GAS.

On Ethereum blockchain, GAS is an execution fee which is used to compensate miners for the computational resources required to power smart contracts. If the network usage is increasing, so will the value of GAS optimization.

These are some of the requirements that must be met to reduce GAS consumption:

- Short-circuiting.
- Remove redundant or dead code.
- Delete unnecessary libraries.
- Explicit function visibility.
- Use of proper data types.
- Use hard coded CONSTANT instead of state variables.
- Avoid expensive operations in a loop.
- Pay special attention to mathematical operations and comparisons.

Duplicate Logic

The logic of the *onlyOwner* modifier in the ***_release*** method of the **TimelockExtenable** contract is executed twice, the first one during the public call to the *release* method or ***releaseAndExtend*** and the second one, during the execution of the internal method itself ***_release***.

It is advisable to check the visibility of this function because changing from internal to private could remove the *onlyOwner* modifier of the ***_release*** with the corresponding gas savings.

```
function release() public virtual onlyOwner {···
}

function releaseAndExtend(uint256 newReleaseTime) public virtual onlyOwner {···
}

function _release() internal onlyOwner {···
}
```

Likewise, the *require* that validates ***_releaseTime*** in the ***releaseAndExtend*** method, is also executed twice, since the aforementioned method already does that check on line 91, as shown below.

```
function releaseAndExtend(uint256 newReleaseTime) public virtual onlyOwner {
    require(
        block.timestamp >= _releaseTime,
        "TokenTimelock: current time is before release time"
    );
    require(
        newReleaseTime > block.timestamp,
        "TokenTimelock: release time is before current time"
    );
    _release();
    _releaseTime = newReleaseTime;
}

function _release() internal onlyOwner {
    // solhint-disable-next-line not-rely-on-time
    require(
        block.timestamp >= _releaseTime,
        "TokenTimelock: current time is before release time"
    );

    uint256 amount = _token.balanceOf(address(this));
    require(amount > 0, "TokenTimelock: no tokens to release");

    _token.safeTransfer(_beneficiary, amount);
}
```

Use of Require statement without reason message

It was verified that the reason message is not specified in some *require* instruction, in order to give the user more information, which consequently makes it more user friendly.

An example of this vulnerability can be found in:
OwnedUpgradeabilityProxy.sol:32

```
) public payable {
    require(_implementation() == address(0));
    InitializableUpgradeabilityProxy.initialize(_logic, _data);
    _setAdmin(_admin);
    emit Initialized(_admin, _logic);
}
```

This functionality is compatible since version *0.4.22* and the contract's pragma indicates *0.7.0*, this will result in compatibility with this feature.

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:

```
// SPDX-License-Identifier: MIT

pragma solidity ^0.7.0;
```

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

References:

- https://github.com/ethereum/solidity/blob/develop/Changelog.md#074-2020-10-19

Invest in Security, invest in your future