Smart Contract Security Audit

eWit

28-05-2021

Content

1. Introduction

eWIT is an ERC-20 token for Ethereum blockchain. The platform is a bridge between Witnet and Ethereum to let users send WIT and receive an equivalent amount of eWIT. Also, a bridge between Ethereum and Witnet to let users send back their eWIT and receive an equivalent amount of WIT.

As requested by eWit 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 eWit Smart Contract source code.

2. Disclaimer

This document only represents the results of the code audit conducted by Red4Sec Cybersecurity and should not be used in any way to make investment decisions or as investment advice on a project.

Likewise, the report should not be considered neither "endorsement" nor "disapproval" of the guarantee of the correct business model of the analyzed project.

3. Scope

The eWit review includes the source code of the smart contract deployed at address `0x8CC8731499849C2d029E9282C611Ce9E9BD99f20` in Rinkeby Ethereum network and its subsequent review of the applied fixes in the Mainnet `0x56EE175FE37CD461486cE3c3166e0CaFCcd9843f`.

4. Conclusions

To this date, 28th of May 2021, the general conclusion resulting from the conducted audit is that eWit's smart contract is secure and do not present any critical-high known vulnerabilities, although Red4Sec has found a few potential improvements.

A few low impact issues were detected and classified only as informative, but they will continue to help eWit to improve and optimize the quality of the project.

It must be mentioned that the eWit contract has control over the minting, the swaps and the fees. However, the verification of the authenticity and legitimacy of the Witnet transactions, their amounts and properties, is done outside of the contract by the WitSwap system.

Find below a complete list of the issues detected during the audit and their current status after completing the remediations review.

Table of vulnerabilities

Id.	Vulnerability	Risk	State
001	Contracts Management Risks	Informative	Assumed
002	Wrong logic in Mint function	Low	Fixed
003	Improvable Swapping functionality	Informative	Partially
004	Lack of Indexed Events	Informative	Fixed
005	GAS Optimizations	Informative	Fixed
006	Outdated Third-Party Libraries	Informative	Fixed
007	Outdated Compiler Version	Informative	Fixed

5. Issues & Recommendations

Contracts Management Risks

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

Admin role

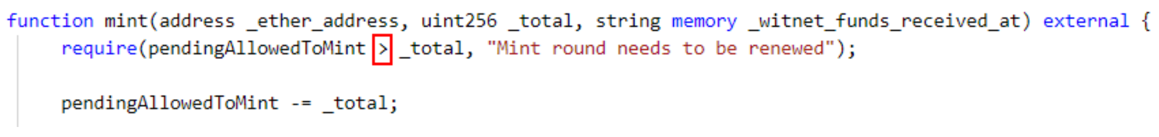
Although the **eWit** contract indicates that the admin's role will be delegated to a multi-signature wallet, it is important to highlight that said role will have control over different values of the contract and will have the possibility of minting arbitrarily.

Possible frozen token

Even though this logic is intentional, it is necessary to mention that the **eWit** token allows to pause its functionality (*transfer, mint, …*) and consequently revoke or give up the administrator role and the pausable role, which would finally leave the token permanently and irrevocably useless.

Wrong logic in Mint function

The mint function of the **eWit** contract verifies that the total to be mined is greater than the amount pending to be mined, *pendingAllowedToMint*, this check should be greater than or equal to it, since otherwise it will never be possible to mine 100% of the pending allowed to be mined.



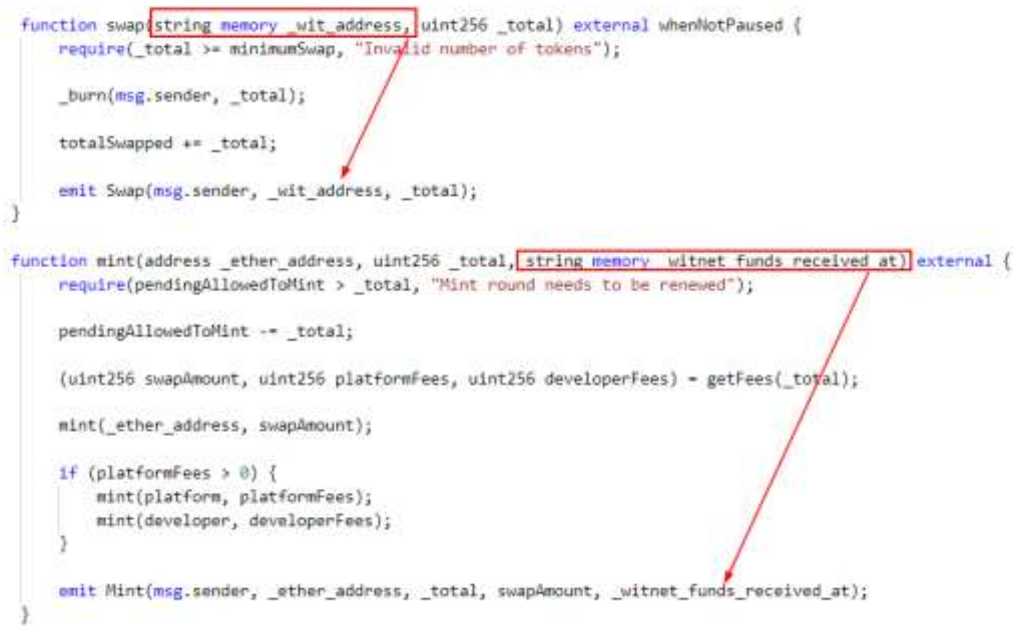
```
function mint(address _ether_address, uint256 _total, string memory _witnet_funds_received_at) external {
    require(pendingAllowedToMint > _total, "Mint round needs to be renewed");

    pendingAllowedToMint -= _total;
```

Improvable Swapping functionality

One of the main functions of the **eWit** contract is to facilitate the swap between the *Witnet* and the *Ethereum* blockchains, for this reason the *swap* and *mint* functions are implemented. Both functions can be improved by adding input verifications and a possible on-chain registry.

First, the *swap* method does not verify the format of *_wit_address* and the *mint* method does not check *_witnet_funds_received_at*, while both could be performing some minimal checks, such as reviewing their sizes, or if applicable, that *_witnet_funds_received_at* is not duplicated.

Secondly, a registry of the mining and the swaps produced can be included, making this information accessible so that third-party contracts or those belonging to our system can consult it.
This will also help to not relegate uniquely in the events, to be able to access the information of the carried-out operations.



```
function swap(string memory _wit_address, uint256 _total) external whenNotPaused {
    require(_total >= minimumSwap, "Invalid number of tokens");

    _burn(msg.sender, _total);

    totalSwapped += _total;

    emit Swap(msg.sender, _wit_address, _total);
}

function mint(address _ether_address, uint256 _total, string memory _witnet_funds_received_at) external {
    require(pendingAllowedToMint > _total, "Mint round needs to be renewed");

    pendingAllowedToMint -= _total;

    (uint256 swapAmount, uint256 platformFees, uint256 developerFees) = getFees(_total);

    mint(_ether_address, swapAmount);

    if (platformFees > 0) {
        mint(platform, platformFees);
        mint(developer, developerFees);
    }

    emit Mint(msg.sender, _ether_address, _total, swapAmount, _witnet_funds_received_at);
}
```

Lack of Indexed Events

Smart contract event indexing can be used to filter during event querying. This can be very useful when making dApps or in the off-chain processing of the events in our contract, as it allows filtering by specific addresses, making it much easier for developers to query the results of invocations.

The Swap and the Mint events of the contract are presumably intended to be consulted by dApps and/or by external systems, so indexing the events will facilitate the work of these systems.

It would be convenient to review the **eWit** contract in order to ensure that all the events have the necessary indexes for the correct functioning of the possible dApps. Addresses are usually the best argument to filter an event.

References

- https://docs.soliditylang.org/en/latest/contracts.html#events

GAS Optimizations

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.

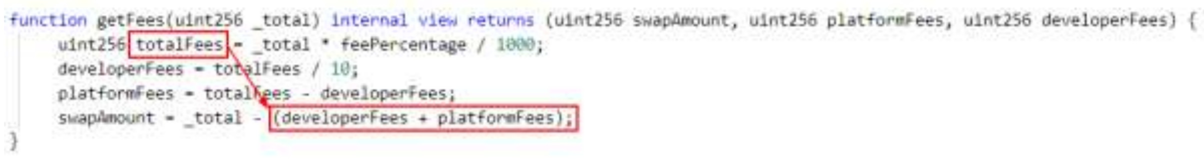
On Ethereum blockchain, GAS is an execution fee which is used to compensate miners for the computational resources required to power smart contracts. If the network usage is increasing, so will the value of GAS optimization.

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.

Variable Optimization

The *getFees* function of the **eWit** contract makes a sum of the developer's fee and of the platform's fee to get the totalFee, when it has been previously calculated, this operation is not necessary and can be avoided with a consequent gas saving.

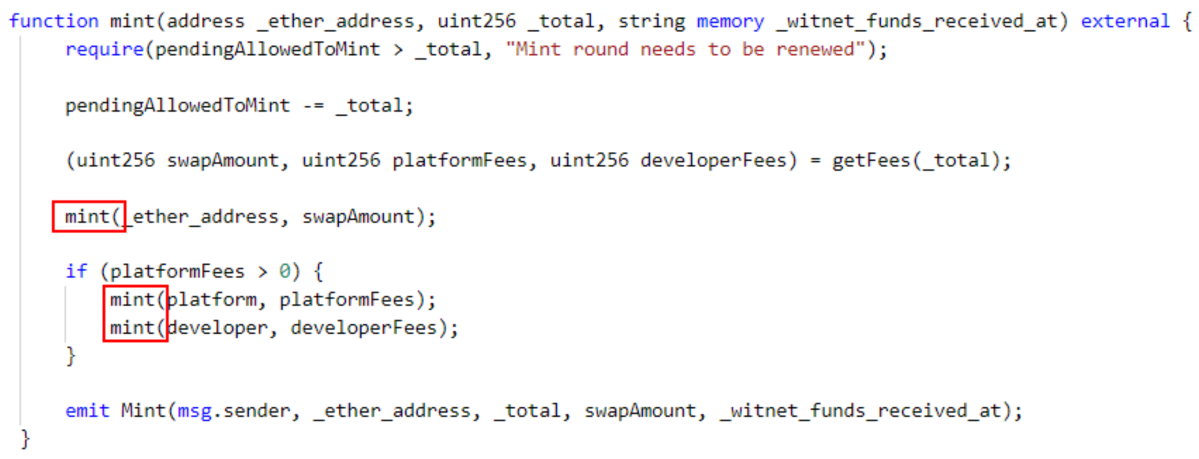


```
function getFees(uint256 _total) internal view returns (uint256 swapAmount, uint256 platformFees, uint256 developerFees) {
    uint256 totalFees = _total * feePercentage / 1000;
    developerFees = totalFees / 10;
    platformFees = totalFees - developerFees;
    swapAmount = _total - (developerFees + platformFees);
}
```

Logic Optimization

The *mint* function of the **eWit** contract does not have an *onlyMinter* modifier, it delegates this verification to the *mint* internal method of the **ERC20PresetMinterPauser** contract. This causes the role of MINTER_ROLE to be checked three times, once for each call to that method.

By moving the logic of this verification to the mint method of the **eWit** contract and by changing the calls to *_mint*, it is possible to save a significant amount of gas for each call to that method.

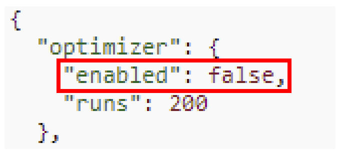


```
function mint(address _ether_address, uint256 _total, string memory _witnet_funds_received_at) external {
    require(pendingAllowedToMint > _total, "Mint round needs to be renewed");

    pendingAllowedToMint -= _total;

    (uint256 swapAmount, uint256 platformFees, uint256 developerFees) = getFees(_total);

    mint(_ether_address, swapAmount);

    if (platformFees > 0) {
        mint(platform, platformFees);
        mint(developer, developerFees);
    }

    emit Mint(msg.sender, _ether_address, _total, swapAmount, _witnet_funds_received_at);
}
```

Compiler Optimization

We must highlight that the code deployed for the **eWit** contract is not compiled with optimizations, enabling the compiler optimizations will save execution GAS. It is useful to enable optimization for the contracts since it will reduce the number of instructions to be executed, which will result in GAS savings.

Optimization Enabled: **No** with **200** runs



```
{
  "optimizer": {
    "enabled": false,
    "runs": 200
  },
```

Executions Cost

The use of constants is recommended as long as the variables are never to be modified. In this case the variables “name”, “symbol” and “decimals” of the eWit contract should be declared as constants since they would not be necessary to access the storage to read the content of these variables and therefore the execution cost is much lower.

Storage Optimization

The use of the *inmutable*[1] keyword is recommended to obtain less expensive executions, by having the same behavior as a constant. However, by defining its value in the constructor we have a significant save of GAS.
This behavior has been observed in:

- ewit.sol: 80 (developer variable)

Outdated Third-Party Libraries

The smart contracts analyzed inherit functionalities from open-zeppelin contracts that have been labeled obsolete and/or outdated; this does not imply a vulnerability by itself, because their logic does not present them, but it does imply that an update is not carried out by third party packages or libraries.

Currently the latest version of Open Zeppelin contracts is *4.1.0* for solidity *0.8* version.

Additionally, these OpenZeppelin contracts are under the MIT license, which requires its license/copyright to be included within the code.

Detected outdated contracts

- AccessControl.sol
- EnumerableSet.so
- ERC20.sol

Recommendations

- Include third-party codes by package manager.

[1] https://docs.soliditylang.org/en/v0.6.5/contracts.html#immutable

- Include in the eWit project any references/copyright to OpenZeppelin code 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 this version.

We have detected that the audited contract uses the pragma *^0.8.0* and it was compiled with version *v0.8.2+commit.661d1103* which seems vulnerable to "ABI Decode Two-Dimensional Array Memory Issue", this issue was fixed on *0.8.4* version.

Solidity branch *0.8* also has important bug fixes in the caching of Keccak-256 hashes up until *0.8.3* version, so it is recommended to use the most up to date version of the pragma.

It is always of good policy to use the most up to date version of the pragma.

References

- https://github.com/ethereum/solidity/blob/develop/Changelog.md
- https://etherscan.io/solcbuginfo?a=ABIDecodeTwoDimensionalArrayMemory
- https://blog.soliditylang.org/2021/03/23/keccak-optimizer-bug

RED4SEC

Invest in Security, invest in your future