Security Audit ERC-20 Smart Contract

Sentivate

2020-08-07

Introduction

Sentivate is a hybrid web, centralized focused but enhanced by decentralized components, built to be a viable & realistic replacement for the modern web. The network is designed to go beyond the capabilities that any solely centralized or decentralized network could offer. Sentivate addresses the following issues directly: bandwidth crisis, outdated protocols, broken DNS, lack of accountability, lack of identity, reactive security, Domain rules, and web categorization.

As requested by Sentivate 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 Sentivate Smart Contract source code.

Scope

The scope of this evaluation includes:

- Description: Sentivate Smart Contract Security Audit.
- Smart Contract:
  - https://etherscan.io/address/0x7865af71cf0b288b4e7f654f4f7851eb46a2b7f8

Conclusions

The general conclusion resulting from the conducted audit, is that the *Sentivate's Smart Contract* is secure and does not present any known vulnerabilities.

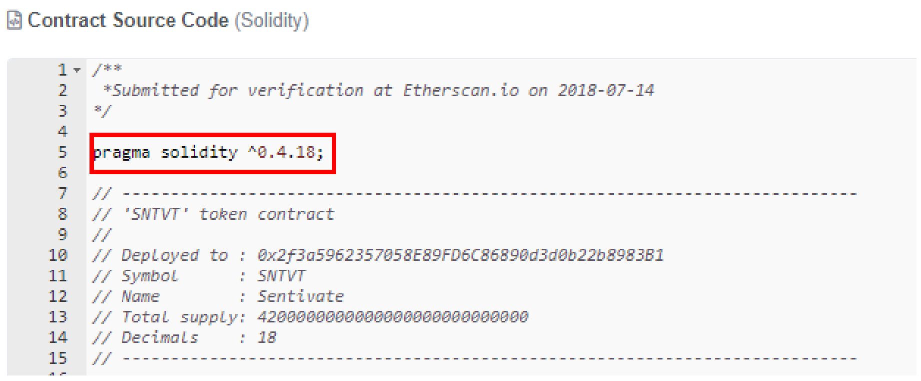
The overall impression about code quality and organization is very positive, although Red4Sec has found some potential improvements, this do not pose any risk by themselves. We have classified such issues as informative only, but it will help Sentivate to continue to improve the security and quality of its developments.

Recommendations

Outdated Compiler Version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 Using an outdated version of the compiler can be problematic, especially if there are errors that have been made public or known vulnerabilities that affect such version.

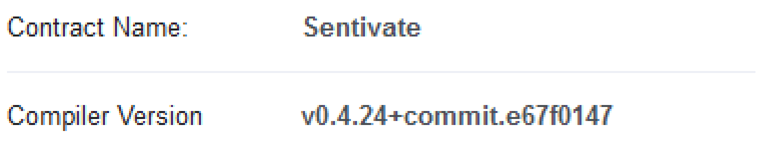
We have detected that the audited contract uses the following version of Solidity *pragma ^0.4.18*:


Contract Source Code (Solidity)

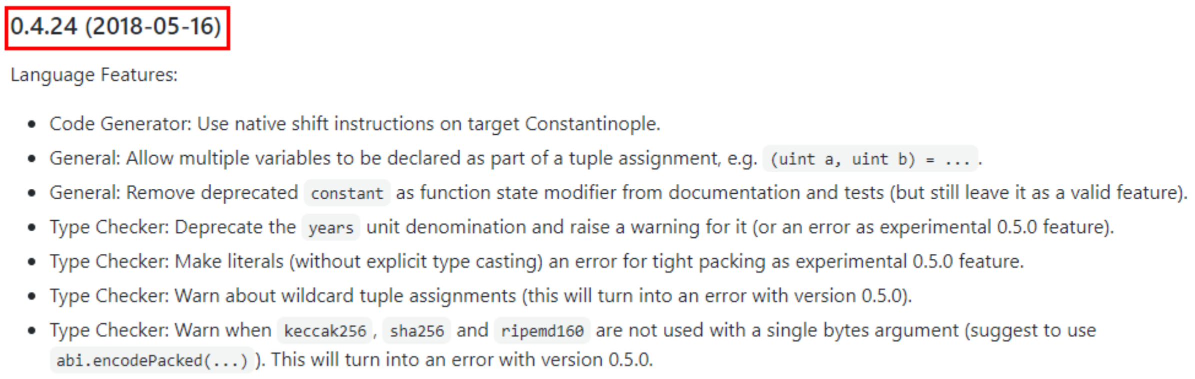
```
/**
 *Submitted for verification at Etherscan.io on 2018-07-14
*/

pragma solidity ^0.4.18;

// ----------------------------------------------------------------------------
// 'SNTVT' token contract
//
// Deployed to : 0x2f3a5962357058E89FD6C86890d3d0b22b8983B1
// Symbol      : SNTVT
// Name        : Sentivate
// Total supply: 4200000000000000000000000000
// Decimals    : 18
// ----------------------------------------------------------------------------
```

Nevertheless, when the deploy was made (14th of July 2018), the last available version was 0.4.24, therefore, the pragma used should have been this one.

Finally, even though the chosen pragma was lower, the contract was compiled with the version **v0.4.24+commit.e67f0147**, as we can observe in the following image.



Contract Name:	Sentivate
Compiler Version	v0.4.24+commit.e67f0147


0.4.24 (2018-05-16)

Language Features:

- Code Generator: Use native shift instructions on target Constantinople.
- General: Allow multiple variables to be declared as part of a tuple assignment, e.g. `(uint a, uint b) = ...`.
- General: Remove deprecated `constant` as function state modifier from documentation and tests (but still leave it as a valid feature).
- Type Checker: Deprecate the `years` unit denomination and raise a warning for it (or an error as experimental 0.5.0 feature).
- Type Checker: Make literals (without explicit type casting) an error for tight packing as experimental 0.5.0 feature.
- Type Checker: Warn about wildcard tuple assignments (this will turn into an error with version 0.5.0).
- Type Checker: Warn when `keccak256`, `sha256` and `ripemd160` are not used with a single bytes argument (suggest to use `abi.encodePacked(...)`). This will turn into an error with version 0.5.0.

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

References

- https://github.com/ethereum/solidity/blob/develop/Changelog.md

GAS Optimization on SafeMath

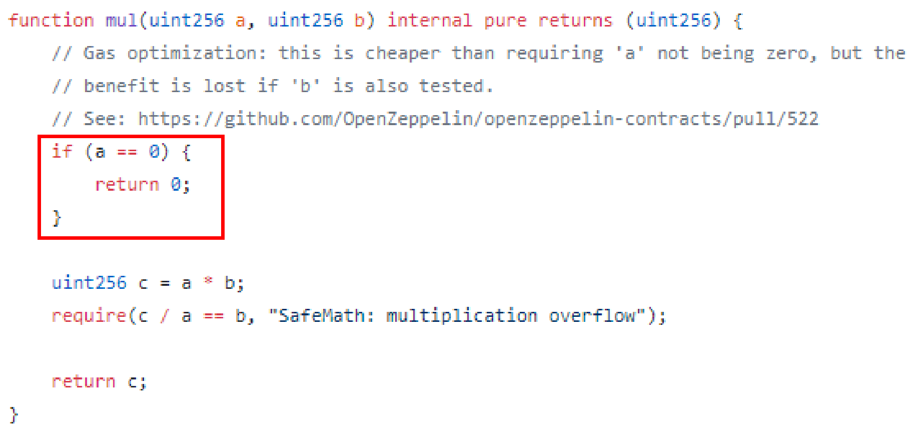
On Ethereum blockchain, gas is an execution fee which is used to compensate miners for the computational resources required to power smart contracts. If the network usage is increasing, so will the value of gas optimization.

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 hardcoded CONSTANT instead of state variables.
- Avoid expensive operations in a loop.
- Pay special attention to mathematical operations and comparisons.

It has been observed that some of these requirements are not met in a few sections of the code. We have verified that function *safeMul* of the contract *SafeMath* can be optimized in order to save GAS.

The creators of the *SafeMath* of OpenZeppelin indicate that is enough, to just check that the variable is not equal to 0, as shown below:



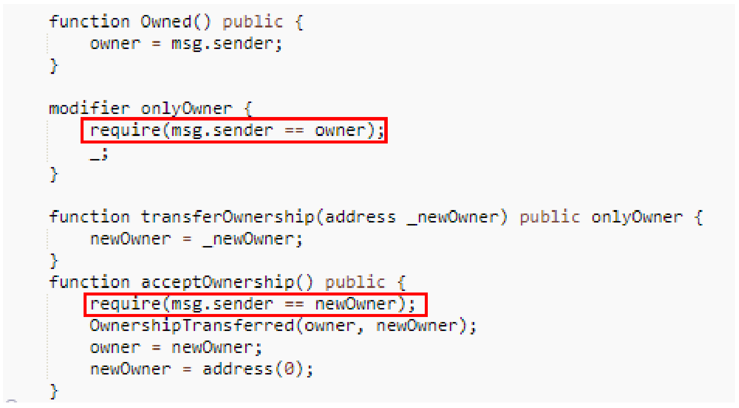
```
function mul(uint256 a, uint256 b) internal pure returns (uint256) {
    // Gas optimization: this is cheaper than requiring 'a' not being zero, but the
    // benefit is lost if 'b' is also tested.
    // See: https://github.com/OpenZeppelin/openzeppelin-contracts/pull/522
    if (a == 0) {
        return 0;
    }

    uint256 c = a * b;
    require(c / a == b, "SafeMath: multiplication overflow");

    return c;
}
```

References

- https://github.com/OpenZeppelin/openzeppelin-contracts/blob/09014f90f9da2b06742101f90dac20aa58e06cc8/contracts/math/SafeMath.sol#L78-L83

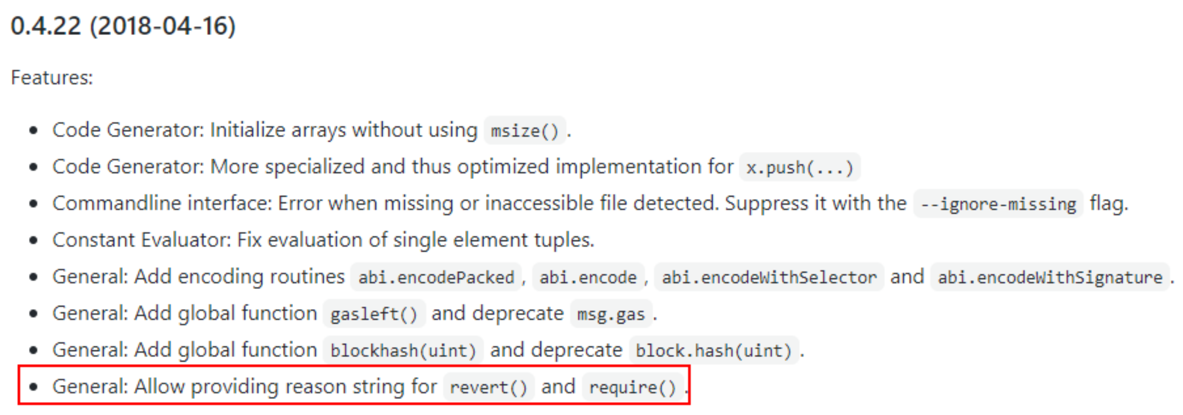
Require message without reason

Throughout the audit, it was verified that the reason message is not specified in the ***require*** *method,* in order to give the user more information, consequently making it more user friendly.



```
function Owned() public {
    owner = msg.sender;
}

modifier onlyOwner {
    require(msg.sender == owner);
    _;
}

function transferOwnership(address _newOwner) public onlyOwner {
    newOwner = _newOwner;
}
function acceptOwnership() public {
    require(msg.sender == newOwner);
    OwnershipTransferred(owner, newOwner);
    owner = newOwner;
    newOwner = address(0);
}
```

This functionality is compatible with the version 0.4.22 and following versions, and even though the contract's pragma indicates an older version, the contract was compilated with 0.4.24 version, which results compatible.



0.4.22 (2018-04-16)

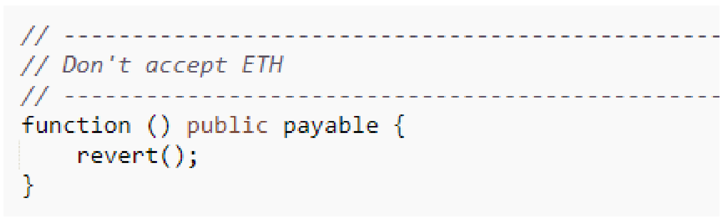
Features:

- Code Generator: Initialize arrays without using `msize()`.
- Code Generator: More specialized and thus optimized implementation for `x.push(...)`
- Commandline interface: Error when missing or inaccessible file detected. Suppress it with the `--ignore-missing` flag.
- Constant Evaluator: Fix evaluation of single element tuples.
- General: Add encoding routines `abi.encodePacked`, `abi.encode`, `abi.encodeWithSelector` and `abi.encodeWithSignature`.
- General: Add global function `gasleft()` and deprecate `msg.gas`.
- General: Add global function `blockhash(uint)` and deprecate `block.hash(uint)`.
- General: Allow providing reason string for `revert()` and `require()`.

The following lines present ***require*** without a message: 24, 27, 32, 35, 82 and 90.

Do not assume that ETH cannot be received

Although the logic of the contract itself does not imply any risk to the fact that it contains or not an amount of Ethereum, it is important to mention that the used protection to avoid receiving Ethereum will only prevent us from receiving human error but never intentionally.



```
// ------------------------------------------------
// Don't accept ETH
// ------------------------------------------------
function () public payable {
    revert();
}
```

There are several ways to send ether on the Ethereum network without triggering the **fallback** method. Just as the **transferAnyERC20Token** method is implemented to transfer ERC20 tokens, it would be worth to implement a similar method for the owner to manage Ethereum.

References

- https://medium.com/@aniketengg/ways-to-send-eth-to-a-contract-having-throw-in-fallback-function-41765db796de