RED4SEC

Smart Contract Security Report

Exeedme

2021-06-04

Content

Introduction

Exeedme is a blockchain-powered tournament platform allowing gamers at all skill-levels, developers and organisers to monetise their skills.

Exeedme's vision is to build a fair and trusted "play-to-earn" platform where gamers can play their favourite games, challenge an opponent, or let Exeedme find one that matches their skill-level and bet on their own victory.

As requested by **Exeedme** and as part of the vulnerability review and management process, **Red4Sec** has been asked to perform a security code audit to **evaluate the security of** its smart contract.

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.

Scope

The scope of this evaluation includes the following Exeedme contracts:

- Exeedme Smart Contract:
    - https://github.com/Exeedme/xed-staking
    - XEDStaking.sol: `2ef8ec8d37918af3bc126ed79601fe56dcd04ebb632d5c67d362f0ac92c65ce5`

Conclusions

To this date, 4th of June 2021, the general conclusion resulting from the conducted audit is that Exeedme's smart contract **is secure and does not present any known vulnerabilities** that could compromise the security of the users. However, there are **some issues** that should be reviewed by the Exeedme team and a few potential improvements that are suggested to apply.

- During the audit it has been detected that the logic implemented in the stacking can be improved.
- It is recommended to review all the logic to confirm that the contract behaves as expected.
- A few low impact issues were detected and classified only as informative, but they will continue to help Exeedme improve the security and quality of its developments.
- The absence of the Unit Test has been detected, during the security review, this is a highly recommended practice that has become mandatory in projects destined to manage large amounts of capital.

Exeedme team has fixed the issues detailed in this report and their current status of the contract after the review done by the Red4Sec team is as follows:

Table of vulnerabilities

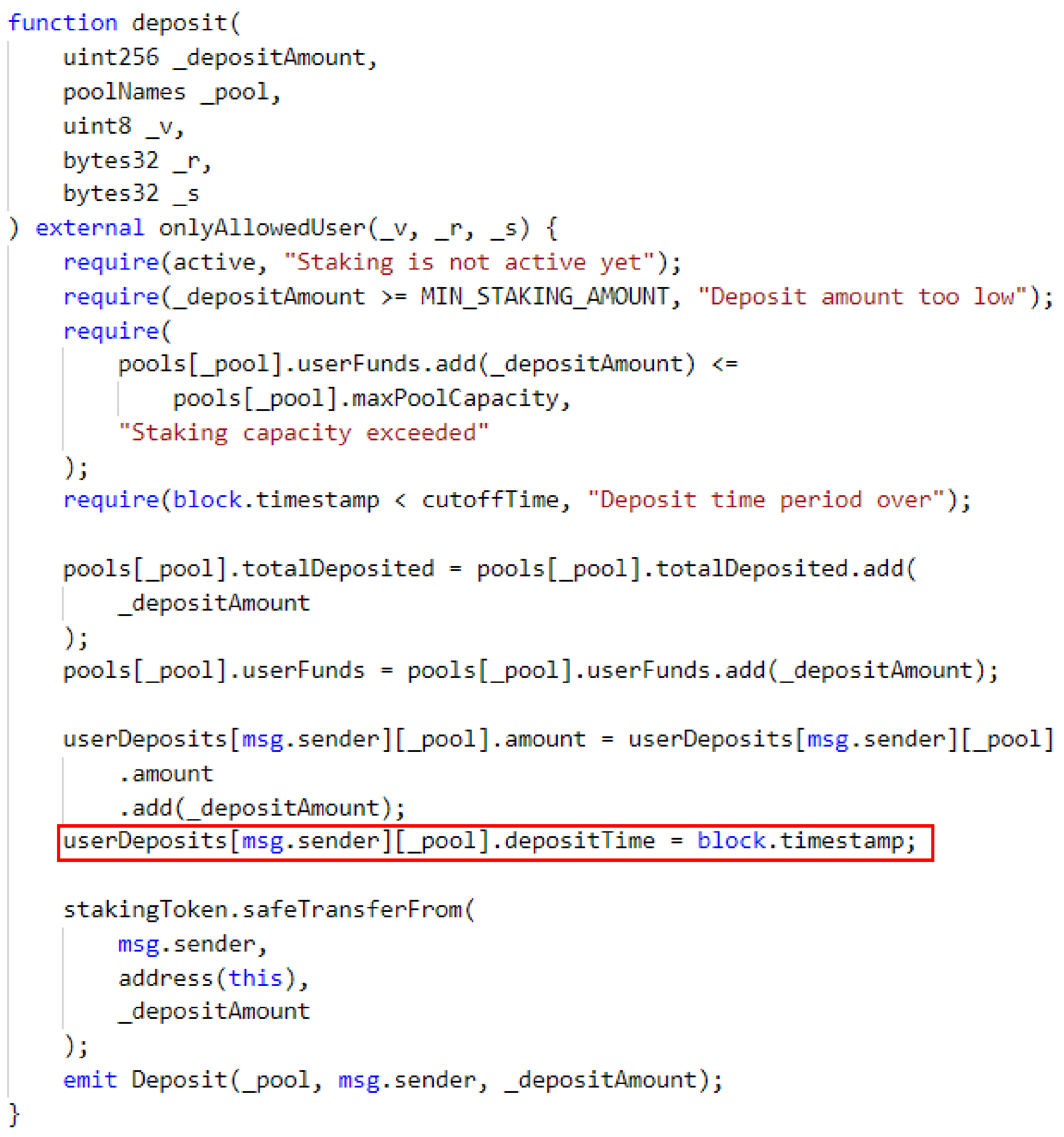
Id.	Vulnerability	Risk	State
EXE01	Improvable Stacking Logic	Low	Assumed
EXE02	Wrong Reward Logic	High	Fixed
EXE03	Improvable allowed user logic	Low	Partially Fixed
EXE04	Decentralization Recommendation	Informative	Fixed
EXE05	Wrong token with fees staking logic	Informative	Intended
EXE06	Lack of Inputs validations	Informative	Assumed
EXE07	Unsecure Ownership Transfer	Informative	Assumed
EXE08	Logic Optimizations	Informative	Fixed
EXE09	Storage Optimizations	Informative	Assumed
EXE10	Reusage storage access	Informative	Assumed
EXE11	Outdated Compiler Version	Informative	Partially Fixed
EXE12	Outdated Third-Party Libraries	Informative	Fixed

Issues & Recommendations

Improvable Stacking Logic

The logic related to the stacking of the users does not correctly contemplate the action of making different deposits in the same pool by the same user.

In the *deposit* function we can see how a user that makes several deposits in the same pool will replace the date of the stacking with the corresponding date of the last deposit.



```
function deposit(
    uint256 _depositAmount,
    poolNames _pool,
    uint8 _v,
    bytes32 _r,
    bytes32 _s
) external onlyAllowedUser(_v, _r, _s) {
    require(active, "Staking is not active yet");
    require(_depositAmount >= MIN_STAKING_AMOUNT, "Deposit amount too low");
    require(
        pools[_pool].userFunds.add(_depositAmount) <=
            pools[_pool].maxPoolCapacity,
        "Staking capacity exceeded"
    );
    require(block.timestamp < cutoffTime, "Deposit time period over");

    pools[_pool].totalDeposited = pools[_pool].totalDeposited.add(
        _depositAmount
    );
    pools[_pool].userFunds = pools[_pool].userFunds.add(_depositAmount);

    userDeposits[msg.sender][_pool].amount = userDeposits[msg.sender][_pool]
        .amount
        .add(_depositAmount);
    userDeposits[msg.sender][_pool].depositTime = block.timestamp;

    stakingToken.safeTransferFrom(
        msg.sender,
        address(this),
        _depositAmount
    );
    emit Deposit(_pool, msg.sender, _depositAmount);
}
```

Therefore, in the *getUserYield* function, its reward is computed as if the entire amount previously stacked had been deposited in the current block. So, if a user that has 99 tokens in stacking for 1 year wants to deposit 1 more token, it would be drastically penalized, since a deposit of 100 tokens would be considered today.

```
function getUserYield(address _userAddress, poolNames _pool)
    public
    view
    returns (uint256)
{
    uint256 depositTime = userDeposits[_userAddress][_pool].depositTime;
    uint256 amount = userDeposits[_userAddress][_pool].amount;

    uint256 daysStaked = (block.timestamp - depositTime) / 1 days;

    uint256 yieldMultiplier = getYieldMultiplier(daysStaked, _pool);
    uint256 daysMultiplier = getNDays(daysStaked, _pool);

    return (amount * yieldMultiplier * daysMultiplier) / (100 * 365);
}
```

Wrong Reward Logic

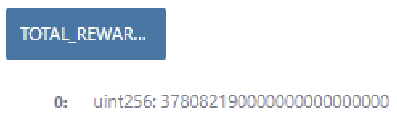
According to the XEDStaking.sol contract, the total reward supply is:

```
uint256 public constant TOTAL_REWARD_SUPPLY = (37808219 * 1 ether) / 100;
```

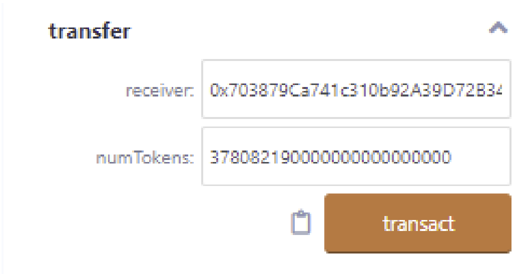
However, it has been possible to detect that by providing the maximum allowed for each pool and by stacking the deposit to the maximum allowed, the contract does not have enough tokens. So, it produces an integer overflow and subsequently a denial of service occurs, which blocks all funds in these pools.

The steps required to replicate this behaviour are as follows:

1) Perform the transfer of the necessary tokens and those that refer to the constant TOTAL_REWARD_SUPPLY:



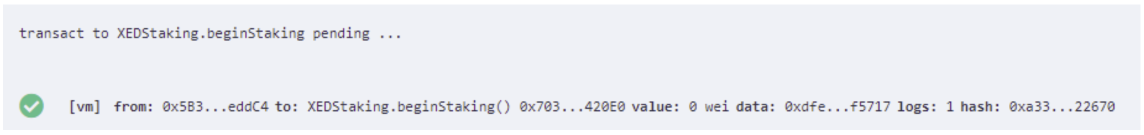
```
0:    uint256: 378082190000000000000000
```


transfer

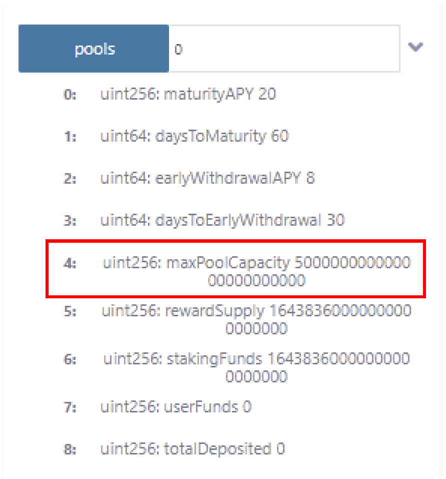
receiver: 0x703879Ca741c310b92A39D72B34

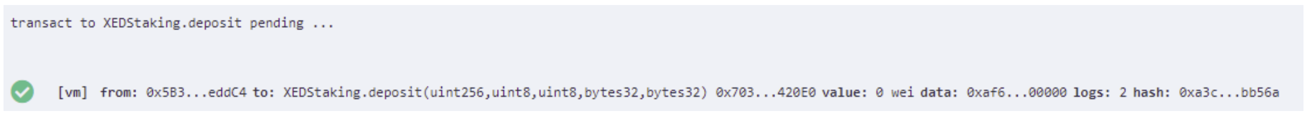
numTokens: 378082190000000000000000

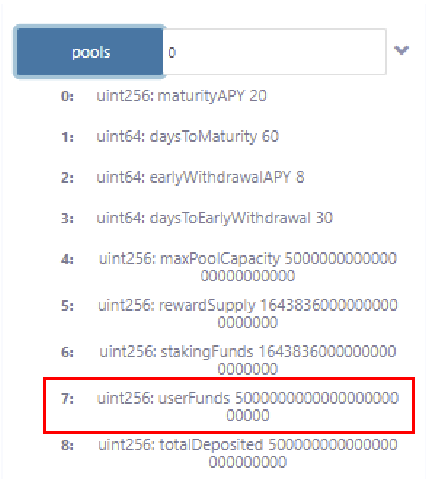
2) Begin the staking through the *beginStaking* method.


```
transact to XEDStaking.beginStaking pending ...

[vm]  from: 0x5B3...eddC4 to: XEDStaking.beginStaking() 0x703...420E0 value: 0 wei data: 0xdfe...f5717 logs: 1 hash: 0xa33...22670
```

3) Fill all the pools to the maximum of their capacity (we repeat this step with pool 1 and 2).

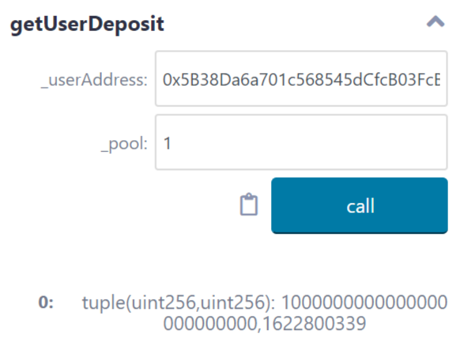

```
pools  0
0:  uint256: maturityAPY 20
1:  uint64: daysToMaturity 60
2:  uint64: earlyWithdrawalAPY 8
3:  uint64: daysToEarlyWithdrawal 30
4:  uint256: maxPoolCapacity 500000000000000000000000
5:  uint256: rewardSupply 16438360000000000000000
6:  uint256: stakingFunds 16438360000000000000000
7:  uint256: userFunds 0
8:  uint256: totalDeposited 0
```


```
transact to XEDStaking.deposit pending ...

[vm]  from: 0x5B3...eddC4 to: XEDStaking.deposit(uint256,uint8,uint8,bytes32,bytes32) 0x703...420E0 value: 0 wei data: 0xaf6...00000 logs: 2 hash: 0xa3c...bb56a
```



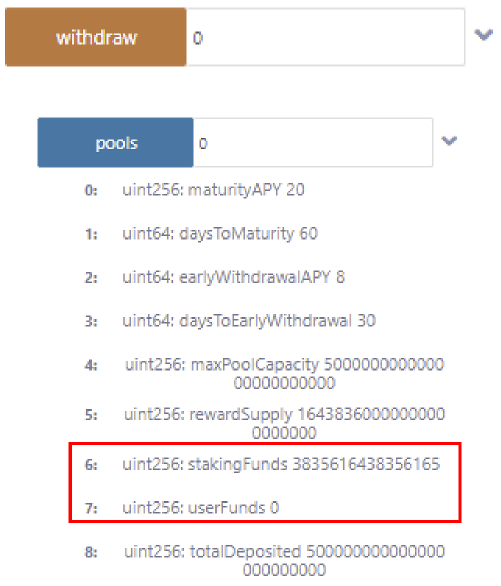
```
0:  uint256: maturityAPY 20
1:  uint64: daysToMaturity 60
2:  uint64: earlyWithdrawalAPY 8
3:  uint64: daysToEarlyWithdrawal 30
4:  uint256: maxPoolCapacity 5000000000000
    00000000000
5:  uint256: rewardSupply 1643836000000000
    0000000
6:  uint256: stakingFunds 1643836000000000
    0000000
7:  uint256: userFunds 5000000000000000000
    00000
8:  uint256: totalDeposited 500000000000000
    000000000
```

4) Wait until the staking of our deposit has reached its maximum values.



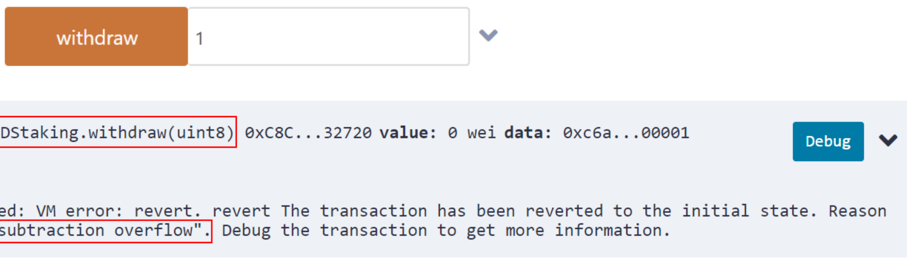
```
_userAddress: 0x5B38Da6a701c568545dCfcB03FcE
_pool: 1

0:  tuple(uint256,uint256): 1000000000000000
    000000000,1622800339
```

5) Make the withdraw from all the pools, making it only possible to withdraw from pool 0, since for the others there are not enough tokens.



```
0:  uint256: maturityAPY 20
1:  uint64: daysToMaturity 60
2:  uint64: earlyWithdrawalAPY 8
3:  uint64: daysToEarlyWithdrawal 30
4:  uint256: maxPoolCapacity 5000000000000
    00000000000
5:  uint256: rewardSupply 1643836000000000
    0000000
6:  uint256: stakingFunds 3835616438356165
7:  uint256: userFunds 0
8:  uint256: totalDeposited 500000000000000
    000000000
```

Following we can see how the funds deposited in the rest of the pools are blocked, since an integer overflow occurs and consequently the aforementioned denial of service of the contract.



```
[vm] from: 0x5B3...eddC4 to: XEDStaking.withdraw(uint8) 0xC8C...32720 value: 0 wei data: 0xc6a...00001
logs: 0 hash: 0x365...9ad22

transact to XEDStaking.withdraw errored: VM error: revert. revert The transaction has been reverted to the initial state. Reason
provided by the contract: "SafeMath: subtraction overflow". Debug the transaction to get more information.
```

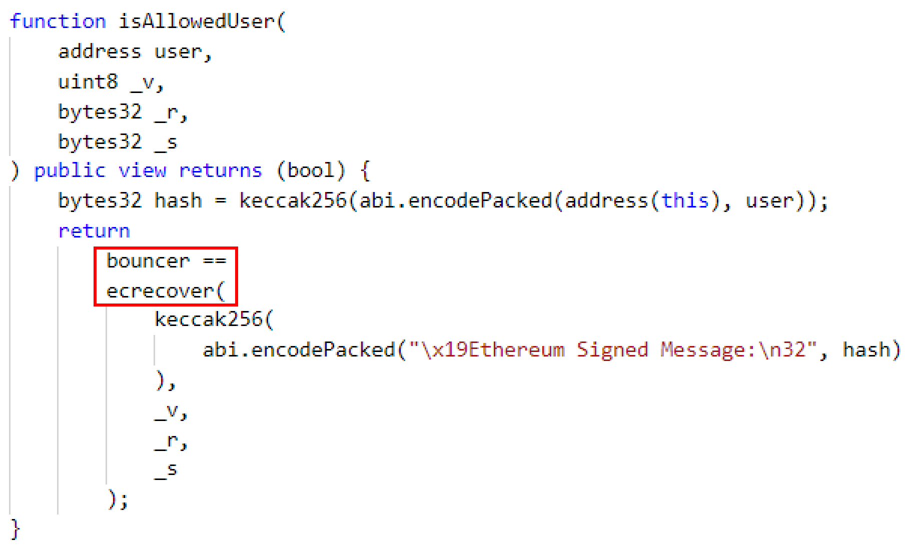
Recommendations

It is recommended to have Unit Tests that verify that the applied logic in the Smart Contract is correct and that all the methods work as expected. Additionally, it is advisable to check that the amounts defined in TOTAL_REWARD_SUPPLY and the values used to calculate the rewards are correct.

Improvable allowed user logic

The logic related to the *onlyAllowedUser* function is not reversible, once the admin grants to a user the ability to make deposits, this capacity cannot be revoked.

It should also be considered that the call to the *ecrecover* method can return *address(0)*[1] when the values are incorrect, so if the administrator decides to set the address of the bouncer to *address(0)*, all the calls that have incorrect values will automatically bypass the signature constraint expected in that modifier.



```
function isAllowedUser(
    address user,
    uint8 _v,
    bytes32 _r,
    bytes32 _s
) public view returns (bool) {
    bytes32 hash = keccak256(abi.encodePacked(address(this), user));
    return
        bouncer ==
        ecrecover(
            keccak256(
                abi.encodePacked("\x19Ethereum Signed Message:\n32", hash)
            ),
            _v,
            _r,
            _s
        );
}
```

Contracts Management Risks

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

Decentralization Recommendation

The *collect* function allows the owner of the contract to drain the tokens held by the **XEDStaking** contract without considering pending withdrawals from the users, which requires trust in the project or in its governance.

However, this call is protected for a period of 220 days after the *cutoffTime* day, so it would only affect the deposits in stacking on that date.

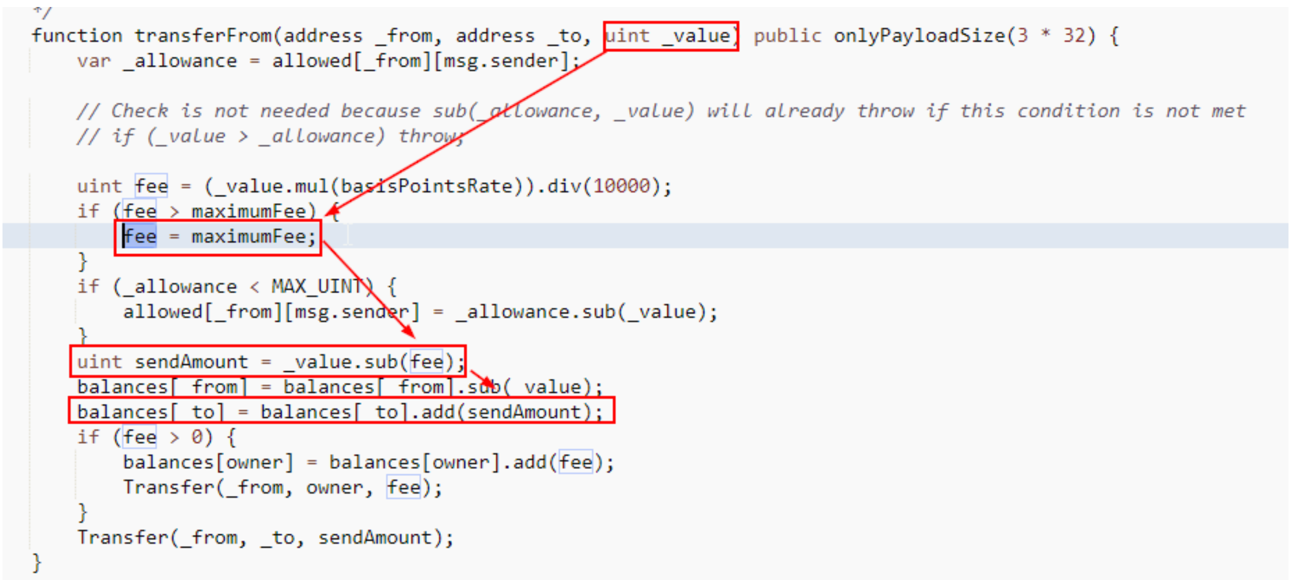
[1] https://github.com/crytic/slither/wiki/Detector-Documentation#missing-zero-address-validation

Wrong Token with fees staking Logic

The current staking logic does not contemplate ERC20 tokens with fee during the *transferFrom*, therefore, the amount received by **XEDStaking** will be less than the expected to carry out the stake.

Some tokens may implement a fee during transfers, this is the case of USDT, even though the project has currently set it to 0. So, the *transferFrom* function would return *'true'* despite receiving less than expected.

https://etherscan.io/address/0xdac17f958d2ee523a2206206994597c13d831ec7#code



```
*/
function transferFrom(address _from, address _to, uint _value) public onlyPayloadSize(3 * 32) {
    var _allowance = allowed[_from][msg.sender];

    // Check is not needed because sub(_allowance, _value) will already throw if this condition is not met
    // if (_value > _allowance) throw;

    uint fee = (_value.mul(basisPointsRate)).div(10000);
    if (fee > maximumFee) {
        fee = maximumFee;
    }
    if (_allowance < MAX_UINT) {
        allowed[_from][msg.sender] = _allowance.sub(_value);
    }
    uint sendAmount = _value.sub(fee);
    balances[_from] = balances[_from].sub(_value);
    balances[_to] = balances[_to].add(sendAmount);
    if (fee > 0) {
        balances[owner] = balances[owner].add(fee);
        Transfer(_from, owner, fee);
    }
    Transfer(_from, _to, sendAmount);
}
```

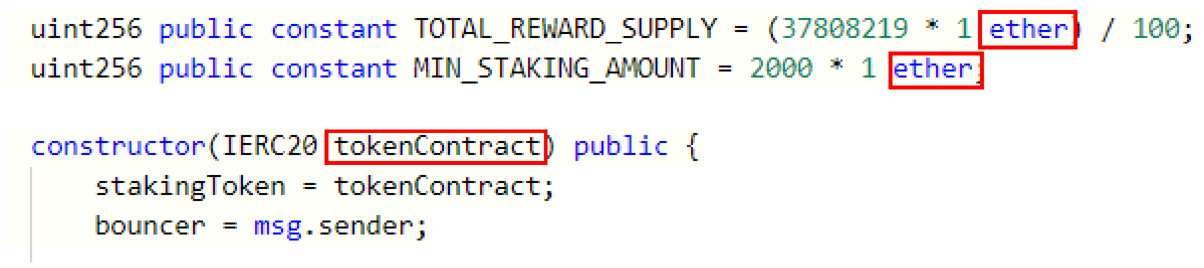
By generating the stake without having said fee, the project will end up assuming the cost of the fee, which may cause economic losses to the project.

Lack of Inputs validations

Some methods of the different contracts in the Exeedme project do not properly check the arguments, which can lead to major errors. Below we list the most significant examples.

- During the creation of the contract, it is not verified that the number of decimals of the ERC20 contract in the *tokenContract* argument is 18. However, there are constants such as *TOTAL_REWARD_SUPPY* or

MIN_STAKING_AMOUNT that assume that the contract will have 18 decimals. It should be considered that not all of the ERC20 contracts have the same number of decimals, as is the case of USDC which is currently of 6 decimals.



```
uint256 public constant TOTAL_REWARD_SUPPLY = (37808219 * 1 ether) / 100;
uint256 public constant MIN_STAKING_AMOUNT = 2000 * 1 ether;

constructor(IERC20 tokenContract) public {
    stakingToken = tokenContract;
    bouncer = msg.sender;
```

Unsecure Ownership Transfer

The modification process of an owner is a delicate process, since the governance of our contract and therefore of the project may be at risk, for this reason it is recommended to adjust the owner's modification logic, to a logic that allows to verify that the new owner is in fact valid and does exist.

Following, we can see a standard logic of the owner's modification where a new owner is proposed first, the owner accepts the proposal and, in this way, we make sure that there are no errors when writing the address of the new owner.

```
function proposeOwner(address _proposedOwner) public onlyOwner
{
        require(msg.sender != _proposedOwner, ERROR_CALLER_ALREADY_OWNER);
        proposedOwner = _proposedOwner;
}

function claimOwnership() public
{
        require(msg.sender == proposedOwner, ERROR_NOT_PROPOSED_OWNER);
        emit OwnershipTransferred(_owner, proposedOwner);
        _owner = proposedOwner;
        proposedOwner = address(0);
}
```

Source reference

- Owable.sol

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.

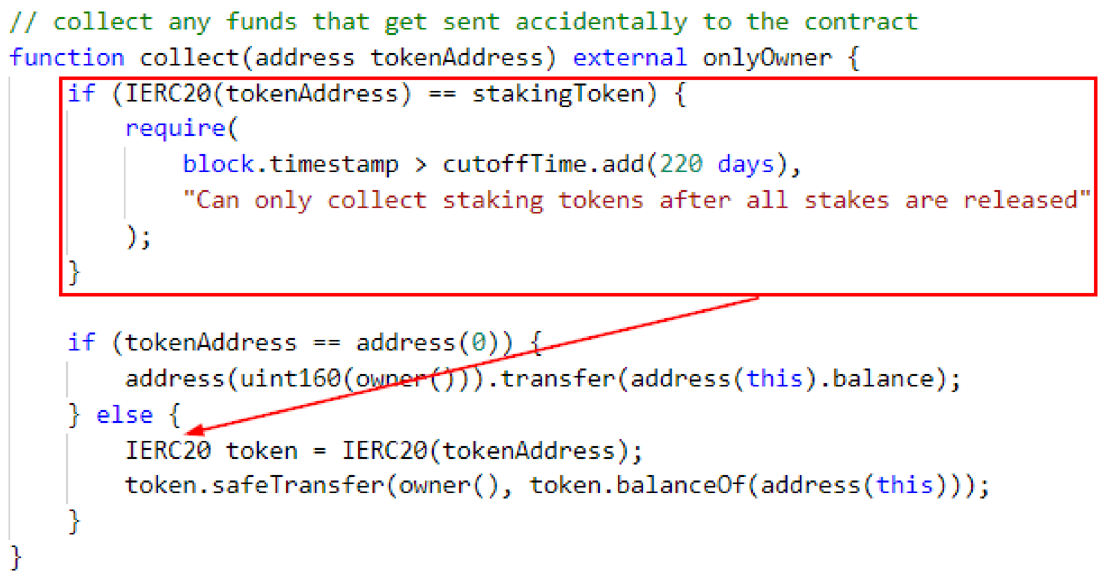
On Ethereum blockchain, GAS is an execution fee which is used to compensate miners for the computational resources required to power smart contracts. If the network usage is increasing, so will the value of GAS optimization.

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.

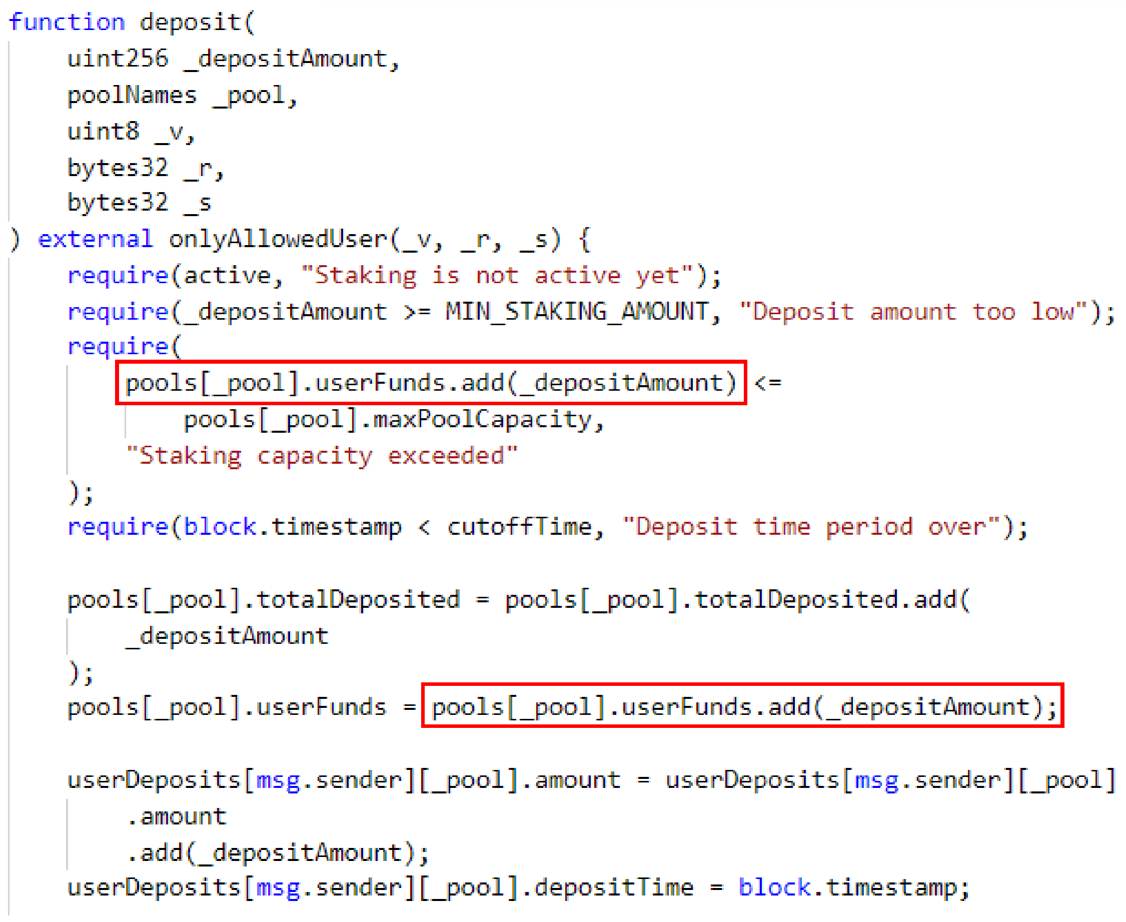
Logic Optimizations

It is possible to optimize the execution of the *collect* method by moving the condition that checks if the token that will be transferred is the *stakingToken* within the condition that has already verified that it was not *address(0)*.



```
// collect any funds that get sent accidentally to the contract
function collect(address tokenAddress) external onlyOwner {
    if (IERC20(tokenAddress) == stakingToken) {
        require(
            block.timestamp > cutoffTime.add(220 days),
            "Can only collect staking tokens after all stakes are released"
        );
    }

    if (tokenAddress == address(0)) {
        address(uint160(owner())).transfer(address(this).balance);
    } else {
        IERC20 token = IERC20(tokenAddress);
        token.safeTransfer(owner(), token.balanceOf(address(this)));
    }
}
```

Additionally, it is possible to reduce the number of operations with its consequent saving of gas in the *deposit* function. If at the moment of adding the total deposited, it is cached in a variable, it will avoid the continuous access to the storage and the *safeMath* operations.



```
function deposit(
    uint256 _depositAmount,
    poolNames _pool,
    uint8 _v,
    bytes32 _r,
    bytes32 _s
) external onlyAllowedUser(_v, _r, _s) {
    require(active, "Staking is not active yet");
    require(_depositAmount >= MIN_STAKING_AMOUNT, "Deposit amount too low");
    require(
        pools[_pool].userFunds.add(_depositAmount) <=
            pools[_pool].maxPoolCapacity,
        "Staking capacity exceeded"
    );
    require(block.timestamp < cutoffTime, "Deposit time period over");

    pools[_pool].totalDeposited = pools[_pool].totalDeposited.add(
        _depositAmount
    );
    pools[_pool].userFunds = pools[_pool].userFunds.add(_depositAmount);

    userDeposits[msg.sender][_pool].amount = userDeposits[msg.sender][_pool]
        .amount
        .add(_depositAmount);
    userDeposits[msg.sender][_pool].depositTime = block.timestamp;
```

Storage Optimizations

The use of the *immutable*[2] keyword is recommended to obtain less expensive executions, by having the same behaviour as a constant. However, by defining its value in the constructor we have a significant save of GAS.

This behaviour has been observed in:

- ERC20.sol:44-46 (*_name* and *_symbol*)

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

Reusage storage access

Through the keyword *storage*[3] it is possible to preserve a pointer to a specific position in the storage, thus preventing access to it and obtaining GAS savings during the execution.

It is possible to obtain GAS savings using this technique in the *deposit* method of the **XEDStaking** contract.

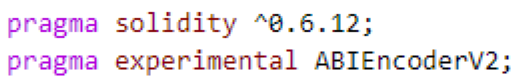
This behaviour has been observed in:

- XEDStaking.sol:167-181

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 following version of Solidity pragma ^0.6.12:



```
pragma solidity ^0.6.12;
pragma experimental ABIEncoderV2;
```

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

The 0.6.12 version of Solc is affected by different known bugs that have already been fixed in later versions. 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

[3] https://docs.soliditylang.org/en/v0.7.4/types.html?highlight=storage#data-location-and-assignment-behaviour

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 therefore it would be convenient to include it as a reference instead of including the sources, in this way we will keep the development environment updated.

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

By using the original sources, in case the project resolves any vulnerability or bug in the code, you would obtain this update automatically. Consequently, avoiding inheriting known vulnerabilities.

Recommendations

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

RED4SEC

Invest in Security, invest in your future