Lack of return value check on transfer and transferFrom

Description

The functions _deposit, _withdraw, and _distributeReward in the xMPH contract use the ERC20 transfer and transferFrom functions without checking that they return true. Some tokens, like BAT, HT (Huobi), and cUSDC, do not revert when transfers fail. Instead, they return false. The lack of a return value check on such tokens could enable users to steal funds. The severity of this issue is informational, as the mph token used in xMPH is likely an instance of MPHToken.sol, which reverts when transfers fail.

Exploit Scenario

A new mph token used in a new deployment of the 88mph protocol does not revert when transfers fail. As a result, users can deposit funds they do not have and then withdraw them, draining funds from the protocol.

Recommendations

Short term, use safeTransfer and safeTransferFrom in xMPH.sol. Long term, always use safeTransfer and safeTransferFrom rather than transfer and transferFrom.

Lack of two-step processforcontract ownership transfers

Description

Many contracts use OpenZeppelin's OwnableUpgradeable contract, whose transferOwnership function transfers contract ownership in a single step. If the owner of a contract were set to an address not controlled by the 88mph team, the contract would be impossible to recover.

Exploit Scenario

Bob, a developer, changes the owner of a contract that inherits from OwnableUpgradeable. By mistake, he passes in the wrong address as the new owner. The development team cannot recover the contract and will have to implement a costly upgrade or redeployment to address the issue.

Recommendations

Short term, replace transferOwnership with a two-step process, as depicted in figure 3.1.

Market makers have a reduced cost for performing front-running attacks

Description

The 0x Protocol 3.0 specification defines how protocol fees are calculated. The protocol fee can be calculated with tx.gasprice * protocolFeeMultiplier, where the protocolFeeMultiplier is an upgradable value meant to target a percentage of the gas used for filling a single order. The suggested initial value for the protocolFeeMultiplier is 150000, which is roughly equal to the average gas cost of filling a single order (thereby doubling the net average cost). Figure 2.1: The protocol fee definition as defined in the 3.0 specification. Market makers receive a portion of the protocol fee for each order filled, and the protocol fee is based on the transaction gas price. Therefore market makers are able to specify a higher gas price for a reduced overall transaction rate, using the refund they will receive upon disbursement of protocol fee pools.

Exploit Scenario

Eve is a market-maker maintaining a distribution pool. Alice submits a profitable transaction to Eve's market. Eve sees the unconfirmed transaction and realizes it will result in a lower overall asset price, and submits a transaction with a higher gas cost and protocol fee, front-running Alice's transaction to sell her asset before the price decreases and increasing her profit from the transaction. Because Eve is a market maker, she receives a portion of the protocol fee she paid to front run Alice's transaction, reducing the overall cost.

Recommendation

Short term, properly document this issue to make sure users are aware of this risk. Establish a reasonable cap for the protocolFeeMultiplier to mitigate this issue. Long term, consider using an alternative fee that does not depend on the tx.gasprice to avoid reducing the cost of performing front-running attacks.

setSignatureValidatorApproval race condition may be exploitable

Exploit Scenario

1. Alice calls setSignatureValidatorApproval(BobContract, True). This allows Bob's contract to validate Alice's signatures.
2. An attacker compromises Bob's contract, so Alice removes the approval calling setSignatureValidatorApproval(Bob, False).
3. The attacker sees Alice’s unconfirmed approval removal and validates a number of malicious transactions or orders before Alice’s transaction is mined.
4. If the attacker's transactions are mined before Alice’s, the malicious transactions or orders can be executed.

Recommendation

Short term, document this behavior to make sure users are aware of the inherent risks of using validators in case of a compromise. Long term, consider monitoring the blockchain using the SignatureValidatorApproval events to catch front-running attacks.

Users cannotspecify a minimum desired interest

Description

DInterest.deposit(amount, maturationTimestamp) computes and returns the amount of interest users will receive at maturationTimestamp. However, users calling this function cannot specify a minimum desired interest to guarantee that the amount of interest will be profitable.

Exploit Scenario

Charlie, a user of the 88mph protocol, calls DInterest.deposit. The amount of interest that is computed and returned is unacceptably low for Charlie. However, he has already deposited his funds into 88mph. To withdraw his funds for use in a more profitable market, Charlie has to pay the early withdrawal fee and the gas costs for another transaction.

Recommendations

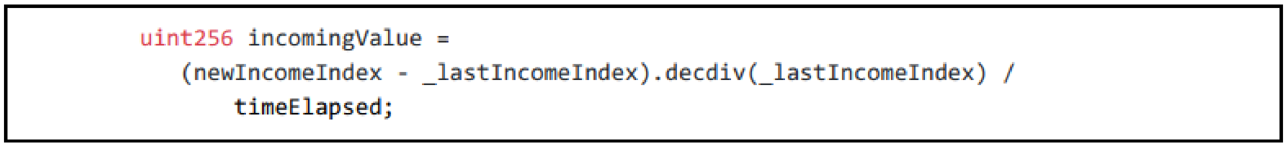
Short term, add a minInterestAmount parameter to DInterest.deposit and insert a require statement—require(interestAmount >= minInterestAmount)—to ensure that the interest amount meets or exceeds the minimum requirement set by the user. Long term, for cases in which a transaction gives a sender an output amount in exchange for an input amount—during a trade, for example—ensure that the sender has control over the minimum desired output amount. Otherwise, state changes outside of the sender’s control, possibly caused by front-running, could result in an unexpectedly low output amount and a loss of funds for the user.

Withdrawing from Yearn to DInterest in a single step can save gas

Linearization of exponential compounding could lead to insolvency

Description

In money markets such as Aave and Compound, liquidity is expected to compound exponentially. However, when computing the money market rate per second, the 88mph protocol assumes that the rate will compound linearly:



```
        uint256 incomingValue =
          (newIncomeIndex - _lastIncomeIndex).decdiv(_lastIncomeIndex) /
              timeElapsed;
```

Figure 6.1: `EMAOracle.sol#L82-L84`

Exploit Scenario

In the example above, the protocol’s estimated yield to the fixed-interest yield minter is much higher than the actual yield will be. As a result, yield token holders (funders) receive a smaller yield than they expected. Furthermore, the protocol incurs a net loss on the deposit, leading to system insolvency.

To compute the true expansion rate per second, the algorithm should take the nth root of the difference of the new income index and the last income index, where n is the number of elapsed seconds. Instead, the algorithm divides the difference by n. We modeled the difference between these two options and found cases in which the current implementation overestimates the ROI. Consider a lending pool that starts with 100 capital units and doubles its capital holdings every two seconds. After four seconds, it will own 400 capital units. The correct prediction would be that the money market will own 800 capital units after two more seconds elapse. However, the current implementation of the algorithm will predict instead that the money market will own 1,000. The consequence of this implementation is that the system could give unreasonably high ROIs to users of fixed-interest yields, potentially increasing the likelihood of system insolvency.

Recommendations

Short term, evaluate the impact of using an arithmetic average rather than a geometric average for computing the compound rate per second. If the impact is severe, consider using the geometric average instead. Alternatively, document the possible consequences of using the arithmetic average, particularly the increased likelihood of a system default.

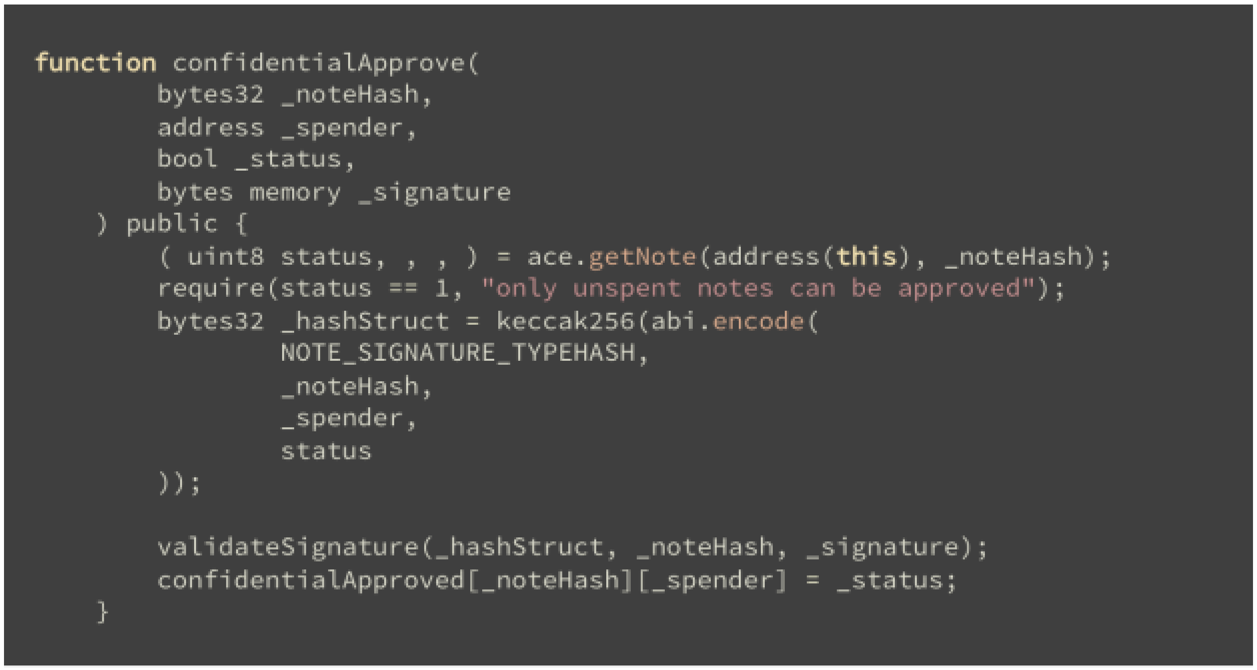
Long term, document any operations in the code that use a simpler algorithm than the mathematical model requires.

https://opentextbc.ca/intermediatealgebraberg/chapter/11-6-compound-interest/#:~:text=An%20application%20of%20exponential%20functions,during%20the%20next%20compounding%20period.

Lack of contract existence check on delegatecall

Replay attack and revocation inversion on confidentialApprove

The confidentialApprove method is used to allow third parties to spend notes on the note owner’s behalf. In the confidentialApprove method, there is a call to validateSignature, which verifies that the signature giving (or revoking) permission to spend is valid and was signed by the owner. However, when verifying the signature, the _status (indicating giving or revoking of permission) is not actually tied to the signature (see figure 13.1), so the same signature used to revoke permission can be used to restore permission. This problem is compounded because there is no mechanism in place to detect the resubmission of previously used signatures.



```
function confidentialApprove(
        bytes32 _noteHash,
        address _spender,
        bool _status,
        bytes memory _signature
    ) public {
        ( uint8 status, , , ) = ace.getNote(address(this), _noteHash);
        require(status == 1, "only unspent notes can be approved");
        bytes32 _hashStruct = keccak256(abi.encode(
                NOTE_SIGNATURE_TYPEHASH,
                _noteHash,
                _spender,
                status
        ));

        validateSignature(_hashStruct, _noteHash, _signature);
        confidentialApproved[_noteHash][_spender] = _status;
    }
```

Figure 13.1: The confidentialApprove method in ZKAssetBase.sol. The _status value is not included in the _hashStruct used to verify the signature.

exploit

This can be exploited to wrongfully give or revoke permissions. Say an owner decides to revoke permission that was previously given to a third party. That party can resubmit either the original signature giving permission or the latest signature revoking permission (where they switch the permission back to true) and wrongfully reinstate their permission to spend notes.

Recommendation

Short term, update confidentialApprove to tie the _status value into the signature used to give/revoke permission to spend. In order for valid signatures to work, this will also require updating signer/index.js in aztec.js to tie this value into the signature (presently, the _status value is always set to true; see Figure 13.2). In addition, we recommend maintaining state to prevent the replay of previous signatures.

Long term, adjust the signature scheme so it is not detachable in this way, in order to prevent similar replay attacks in the future.