Making Web3 Space Safer for Everyone

KALOS

# Meter Liquid Staking

## Security Assessment

Published on : 11 July. 2023
Version v1.1

# Security Report Published by KALOS

v1.1 11 July. 2023

Auditor : Jade, Hun

hojung han hun

## Found issues

| Severity of Issues | Findings | Resolved | Acknowledged | Comment |
|---|---|---|---|---|
| Critical | 3 | 3 | - | - |
| High | 4 | 4 | - | - |
| Medium | 0 | - | - | - |
| Low | 0 | - | - | - |
| Tips | 3 | 2 | - | 1 |

# TABLE OF CONTENTS

# ABOUT US

## Making Web3 Space Safer for Everyone

KALOS is a flagship service of HAECHI LABS, the leader of the global blockchain industry. We bring together the best Web2 and Web3 experts. Security Researchers with expertise in cryptography, leaders of the global best hacker team, and blockchain/smart contract experts are responsible for securing your Web3 service.

Having secured $60B crypto assets on over 400 main-nets, Defi protocols, NFT services, P2E, and Bridges, KALOS is the only blockchain technology company selected for the Samsung Electronics Startup Incubation Program in recognition of our expertise. We have also received technology grants from the Ethereum Foundation and Ethereum Community Fund.

Inquiries: [audit@kalos.xyz](mailto:audit@kalos.xyz)
Website: [https://kalos.xyz](https://kalos.xyz)

# Executive Summary

### Purpose of this report

This report was prepared to audit the security of the project developed by the Meter team. KALOS conducted the audit focusing on whether the system created by the Meter team is soundly implemented and designed as specified in the published materials, in addition to the safety and security of the project.

In detail, we have focused on the following

- Denial of Service
- Freezing of User Assets
- Theft of User Assets
- Function Access Control
- Yield calculation manipulate
- Unhandled Exceptions

### Codebase Submitted for the Audit

The codes used in this Audit can be found on GitHub (https://github.com/meterio/sys-contracts).

The commit of the code used for this Audit is "32b9693224a78a8e11ab4603beb64699ed58b75f",

The commit of the code used for this Audit Review is "ef959429ed1c1693d1527835320d6ef8b67e0e8d",

### Audit Timeline

| Date | Event |
|---|---|
| 2023/05/25 | Audit Initiation (Meter Liquid Staking) |
| 2023/06/09 | Delivery of v1.0 report. |
| 2023/07/11 | Delivery of v1.1 report. |

## Findings

KALOS found 3 Critical, 4 High, 3 Tips severity issues.

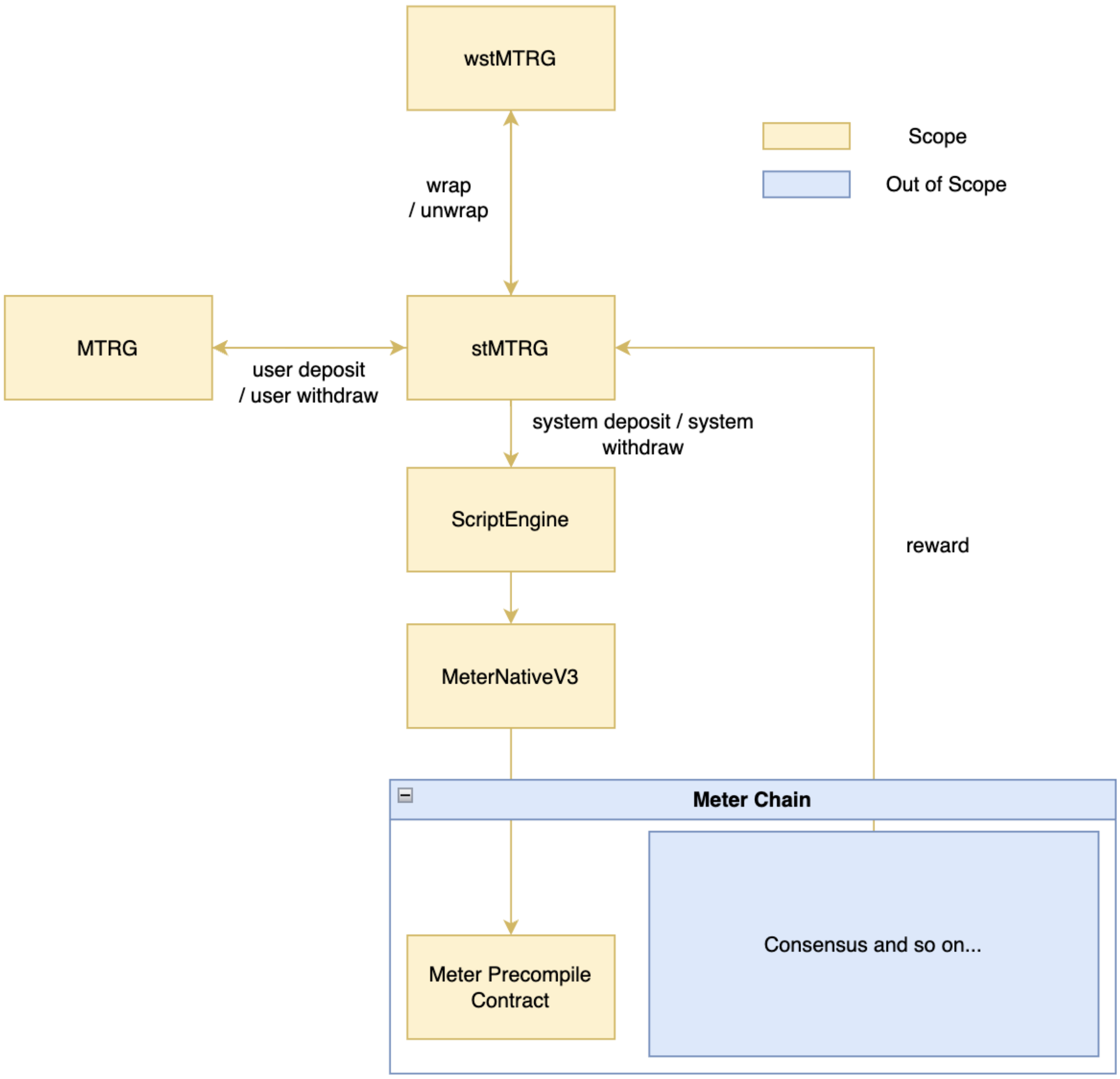
| Severity | Issue | Status |
|---|---|---|
| **Tips** | Using a vulnerable signature verification library | (Resolved - v1.1) |
| **Tips** | Centralization Risk - 1 | (Comment - v1.1) |
| **High** | Lack of Bucket Owner Verification in Updating Bucket Candidates | (Resolved - v1.1) |
| **Critical** | Unwithdrawn Users Cannot Claim MTRG Repayment upon Bucket Closure | (Resolved - v1.1) |
| **Critical** | Asset Loss during Conversion from WstMTRG to stMTRG upon Bucket Closure | (Resolved - v1.1) |
| **Critical** | Potential Unintended Backdoor Vulnerability in MTRG Token | (Resolved - v1.1) |
| **High** | Lack of Blacklist Verification for Deposits and Withdrawals in stMTRG | (Resolved - v1.1) |
| **Tips** | Centralization Risk - 2 | (Resolved - v1.1) |
| **High** | BucketUpdateCandidate does not update candidate's buckets | (Resolved - v1.1) |
| **High** | Unfair Treatment of MTRG Stakers Due to Price Disparity with MTR | (Resolved - v1.1) |

## Remarks

We also analyzed the precompile contract and some components of the meter chain related to meter liquid staking, but the meter chain was outside the scope of our audit.

# OVERVIEW

## Protocol overview



• **stMTRG.sol**

This smart contract implements a staking mechanism for MTRG tokens, enabling users to deposit their MTRG tokens and receive stMTRG tokens in return.

These stMTRG tokens can be later redeemed for MTRG.

The contract is initialized by an administrator who deposits an initial amount of MTRG into a smart contract bucket managed by a script engine.

The script engine is responsible for monitoring and executing various transactions related to the bucket funds. These transactions include opening new buckets, depositing and withdrawing funds, and closing buckets.

Once the contract is initialized, users have the ability to deposit their MTRG tokens into the contract. In return, they receive stMTRG tokens proportionate to their share of the total staked MTRG.

The contract also offers additional features such as the ability to pause and unpause deposits and transfers, establish a blacklist to prevent specific users from transferring tokens, and assign a new candidate address to manage the smart contract bucket.

An admin can completely terminate a bucket associated with a smart contract.

• **WstMTRG.sol**
The WstMTRG contract is an implementation of the ERC20PermitUpgradeable contract standard, designed as a wrapped token contract.

Its primary purpose is to allow users to convert their staked MTRG tokens, which earn staking rewards, into wstMTRG tokens.

These wstMTRG tokens can then be unwrapped back to their original staked MTRG tokens. The amount of wstMTRG tokens received upon wrapping is equivalent to the value of staked MTRG held by the user.

Additionally, the contract provides functionality for users to check the exchange rate between stMTRG and wstMTRG tokens, facilitating transparency in token conversion.

To wrap MTRG tokens into wstMTRG, users are required to first approve the WstMTRG contract to transfer their MTRG tokens.

This approval is necessary for the contract to access and convert the specified amount of MTRG tokens into equivalent wstMTRG tokens.

Conversely, when users wish to unwrap their wstMTRG tokens and receive MTRG tokens in return, they must possess at least the corresponding amount of wstMTRG tokens.

The unwrapping process is initiated by calling the unwrap function.

• **ScriptEngine.sol**
The ScriptEngine smart contract is designed to work in conjunction with the MeterNativeV3 contract, which manages buckets of a native token called MTRG.

This contract offers various functions for efficient bucket management.

The 'bucketOpen' function allows users to create a new bucket associated with their address (msg.sender).

Additionally, it enables users to delegate for a candidate by transferring a specified amount of MTRG tokens to the candidate's address.

With the 'bucketDeposit' function, users can add more MTRG tokens to an existing bucket owned by them.

This function requires input parameters specifying the amount of tokens and the bucket ID.

For withdrawing MTRG tokens from a bucket and creating a new sub-bucket under a recipient address, the 'bucketWithdraw' function is used.

Users provide the amount of tokens to withdraw and the ID of the target bucket.

To close a bucket completely and withdraw all the MTRG tokens, the 'bucketClose' function is employed.

After the maturity time for the bucket has expired, the tokens are transferred to the owner.

The 'bucketUpdateCandidate' function allows users to update the candidate associated with their bucket.

Lastly, the 'boundedMTRG' function is a view function that provides information about the locked amount of MTRG tokens owned by the sender.

It does not modify the contract's state.

Overall, the ScriptEngine smart contract serves as a vital component for managing buckets of MTRG tokens and enables users to perform essential actions such as opening, depositing, withdrawing, closing buckets, and updating candidates.

• **MeterNativeV3.sol**
The MeterNativeV3 contract is an implementation of the IMeterNativeV3 interface and inherits from the NewMeterNative contract.

It facilitates the creation, deposit, withdrawal, and closure of funds held within specific buckets.

These buckets are associated with a particular address and can be voted towards a candidate address.

The `native_bucket_open()` function allows the `owner` to create a new bucket by specifying the desired `amount` of funds from their balance.

The bucket is then associated with the provided `candidateAddr`.

This function returns a new bucket ID and an optional error message if any issues arise.

Using the `native_bucket_deposit()` function, the `owner` can add additional funds to a designated bucket identified by its `bucketID`.

The function utilizes the funds from the `owner`'s balance.

Similarly, an error message is returned if any problems occur during the operation.

For withdrawing funds from a specific `bucketID`, the `native_bucket_withdraw()` function deducts the specified `amount` from the designated bucket.

The withdrawn funds will be sent to the `recipient` after the bucket's `matureTime`.

The function returns a sub `bucketID` and an error message if applicable.

The `native_bucket_close()` function is used to close the designated `bucketID`. After the bucket's `matureTime`, the `owner` will receive their funds.

Any relevant error message will be returned.

Lastly, the `native_bucket_update_candidate()` function enables the update of the `candidateAddr` associated with a specific `bucketID` to a new address.

It returns an error message if any issues arise during the process.

• **Meter-POV chain**
The Meter chain is an EVM-compatible blockchain that combines Proof of Work (PoW) and Proof of Stake (PoS) through a consensus mechanism known as Proof of Value (PoV).
It supports the execution of EVM contracts and includes separate precompiled contracts for Liquid Staking.
In addition to the precompiled contracts, the chain incorporates built-in functionality such as the auction handler, staking handler, and account lock handler, which can be utilized without going through the EVM.

## Scope

```
├── IMeterNativeV3.sol
├── IMeternative.sol
├── IScriptEngine.sol
├── IStMTRG.sol
├── MeterNativeV3.sol
├── NewMeterNative.sol
├── ScriptEngine.sol
├── StMTRG.sol
├── WstMTRG.sol
```

## Roles

The Meter Liquid Staking ecosystem involves various entities that play essential roles in the system's operations and governance.
This overview focuses on three key participants: General Users, Node Operators, and the Admin.
Each entity contributes to the functionality and success of Meter Chain's liquid staking platform.

General Users - General users can participate in Meter Chain's liquid staking and engage in various financial activities within the Meter Chain using the received wstMTRG and stMTRG tokens.

Node Operators - Node Operators, designated by the admin, stake MTRG tokens received through the stMTRG Contract to participate in the consensus process. They earn interest income and distribute it back to the users participating in liquid staking through stMTRG. Node Operators also receive a portion of the fees.

Admin - The admin is responsible for managing the Liquid Staking Dapp. They have the authority to temporarily pause the Dapp, replace the Node Validator delegated with the Contract's assets, update the candidate address, add someone to the blacklist, close the bucket, and permanently block Liquid Staking for the app.

# FINDINGS

## 1. Using a vulnerable signature verification library

|  |  |
|---|---|
| ID: METER-LSD-01 | Severity: Tip |
| Type: Logic Error | Difficulty: - |

### Issue

There is no real security threat, but a signature malleability vulnerability could be caused if the code is updated later.

```
function tryRecover(bytes32 hash, bytes memory signature)
        internal
        pure
        returns (address, RecoverError)
    {

        if (signature.length == 65) {
            bytes32 r;
            bytes32 s;
            uint8 v;
            // ecrecover takes the signature parameters, and the only way to get them
            // currently is to use assembly.
            assembly {
                r := mload(add(signature, 0x20))
                s := mload(add(signature, 0x40))
                v := byte(0, mload(add(signature, 0x60)))
            }
            return tryRecover(hash, v, r, s);
        } else if (signature.length == 64) {
            bytes32 r;
            bytes32 vs;
            // ecrecover takes the signature parameters, and the only way to get them
            // currently is to use assembly.
            assembly {
                r := mload(add(signature, 0x20))
                vs := mload(add(signature, 0x40))
            }
            return tryRecover(hash, r, vs);
        } else {
            return (address(0), RecoverError.InvalidSignatureLength);
        }
    }
```

https://github.com/meterio/sys-contracts/blob/32b9693224a78a8e11ab4603beb64699ed58b75f/contracts/ECDSA.sol#L57-L90

The above code has already been reported as vulnerable in Openzepplin.
(https://github.com/OpenZeppelin/openzeppelin-contracts/security/advisories/GHSA-4h98-2769-gh6h)

Depending on the implementation of the code, Existing signatures can be reused one more.

**Recommendation**

We recommend using the latest version of the ECDSA Library.

**Fix Comment**

ECDSA.sol has been updated to v4.8. Only 65-bytes signature is allowed.

## 2. Centralization Risk

| | |
|---|---|
| ID: METER-LSD-02 | Severity: Tip |
| Type: Logic Error | Difficulty: - |

**Issue**

As a single point of failure, if a single candidate is jailed, users will not be able to earn interest.

**Recommendation**

We recommend associating multiple Candidates with an LSD Contract.

**Fix Comment**

User's tokens are deposited to multiple candidates in turn. However, there is still a possibility of a single point of failure due to staking concentration on a single candidate. We recommended distributing staking uniformly to mitigate this issue. However, they said that the admin of the contract will move the votes from one candidate, and they will balance the votes among the candidates for time to time.

In addition, we found other issues. One issue is that stMTRG adds value to the candidate in jail during this fix review, we recommended adding validation logic to prevent some bucket operations from adding value to the jailed candidate. They will fix this issue.

The other issue is that _transfer() and _withdraw() allow a user to transfer or withdraw an amount even if they have fewer shares than the specified amount. They said that this is an intentional design.

## 3. Lack of Bucket Owner Verification in Updating Bucket Candidates

ID: METER-LSD-03
Type: Logic Error

Severity: High
Difficulty: Low

**Issue**

A malicious user can change the candidates of different buckets on the mainnet.

This issue allows a malicious user to steal all the Block Rewards.

```
{"native_bucket_update_candidate", func(env *xenv.Environment) []interface{} {
                      var args struct {
                             Owner            meter.Address
                             BucketID         meter.Bytes32
                             NewCandidateAddr meter.Address
                      }
                      env.ParseArgs(&args)
                      env.UseGas(meter.GetBalanceGas)
                      err :=
MeterTracker.Native(env.State()).BucketUpdateCandidate(args.Owner, args.BucketID,
args.NewCandidateAddr)
                      if err != nil {
                             return []interface{}{err.Error()}
                      }

                      return []interface{}{""}
               }},
```

https://github.com/meterio/meter-pov/blob/f3c63e474005266d2ec5d4ac4392b965dc62dd84/builtin/meter_tracker_native.go#L330-L345

```
func (e *MeterTracker) BucketUpdateCandidate(owner meter.Address, id meter.Bytes32,
newCandidateAddr meter.Address) error {
        candidateList := e.state.GetCandidateList()
        bucketList := e.state.GetBucketList()

        b := bucketList.Get(id)
        if b == nil {
               return errBucketNotListed
        }
        nc := candidateList.Get(newCandidateAddr)
        if nc == nil {
                return errCandidateNotListed
        }

        c := candidateList.Get(b.Candidate)
        // subtract totalVotes from old candidate
        if c != nil {
                if c.TotalVotes.Cmp(b.TotalVotes) < 0 {
                        return errNotEnoughVotes
                }
                c.TotalVotes.Sub(c.TotalVotes, b.TotalVotes)
        }
        // add totalVotes to new candidate
        nc.TotalVotes.Add(nc.TotalVotes, b.TotalVotes)
        b.Candidate = nc.Addr

        e.state.SetBucketList(bucketList)
        e.state.SetCandidateList(candidateList)
        return nil
}
```

https://github.com/meterio/meter-pov/blob/f3c63e474005266d2ec5d4ac4392b965dc62dd84/builtin/metertracker/meter_tracker_staking.go#L267-L296

The code should only allow the bucket's owner to modify the candidate, but there is no code to check if the transaction sender is the bucket owner.

Therefore, it was possible to modify the candidate of every existing bucket and steal all the block rewards.

### Recommendation

We recommend adding code to check if the Transaction Sender is the Bucket Owner.

### Fix Comment

It checks if the owner of the bucket to be updated is the same as the passed ‘owner’ parameter that is msg.sender who called ScriptEngine.

## 4. Unwithdrawn Users Cannot Claim MTRG Repayment upon Bucket Closure

ID: METER-LSD-04
Type: Logic Error

Severity: Critical
Difficulty: Low

### Issue

Users of stMTRG may not be able to withdraw in the worst-case scenario.

```
function requestClose() public onlyRole(DEFAULT_ADMIN_ROLE) {
        isClosed = true;
        closeTimestamp = block.timestamp;
        emit RequestClost(closeTimestamp);
    }
...
function executeClose() public onlyRole(DEFAULT_ADMIN_ROLE) {
        require(isClosed, "closed!");
        require(
            block.timestamp >= closeTimestamp + CLOSE_DURATION,
            "CLOSE_DURATION!"
        );
        scriptEngine.bucketClose(bucketID);
        emit ExecuteClost(block.timestamp);
    }
```

https://github.com/meterio/sys-contracts/blob/32b9693224a78a8e11ab4603beb64699ed58b75f/contracts/StMTRG.sol#L95-L109

The requestClose and executeClose functions above close the bucket and withdraw all staked MTRG Token.

There needs to be a code in these functions to distribute the withdrawn MTRG to the stMTRG share owners properly.

There is also no code in the precompile contract code to distribute the stMTRG to the stMTRG share owners properly.

```
func (e *MeterTracker) BucketClose(owner meter.Address, id meter.Bytes32, timestamp uint64)
error {
       bucketList := e.state.GetBucketList()

       b := bucketList.Get(id)
       if b == nil {
              return errBucketNotListed
```

```
        }

        if b.Unbounded {
                return errBucketAlreadyUnbounded
        }

        if b.Owner != owner {
                return errBucketNotOwned
        }

        if b.IsForeverLock() {
                return errNoUpdateAllowedOnForeverBucket
        }

        // sanity check done, take actions
        b.Unbounded = true
        b.MatureTime = timestamp + meter.GetBoundLocktime(b.Option) // lock time

        e.state.SetBucketList(bucketList)
        return nil
}
```

https://github.com/meterio/meter-pov/blob/f3c63e474005266d2ec5d4ac4392b965dc62dd84/builtin/metertracker/meter_tracker_staking.go#L117-L143

## Recommendation

Refer to the code below to modify the project code.

This code considers users who want to withdraw after executeClose using the isTerminal and TerminalTimestamp variables.

```
    uint256 TerminalTimestamp = 0;
     bool isTerminal = false;
     function executeClose() public onlyRole(DEFAULT_ADMIN_ROLE) {
         require(isClosed, "closed!");
         require(
             block.timestamp >= closeTimestamp + CLOSE_DURATION,
             "CLOSE_DURATION!"
         );
         scriptEngine.bucketClose(bucketID);
         isTerminal = true;
         TerminalTimestamp = block.timestamp + 1 weeks; // Modify 1weeks based on bound lock
time
         emit ExecuteClost(block.timestamp);
     }

     function _valueToShare(uint256 _value) private view returns (uint256) {
         if(block.timestamp > TerminalTimestamp && isTerminal)
             return
_value.rayMul(_totalShares).rayDiv(MTRG.balanceOf(address(this))).rayToWad();
         return _value.rayMul(_totalShares).rayDiv(totalSupply()).rayToWad();
     }

    function _withdraw(
         address account,
         uint256 amount,
         address recipient
     ) private {
         uint256 burnShares = _valueToShare(amount);
         uint256 accountShares = _shares[account];
         require(
             accountShares >= burnShares,
             "ERC20: burn amount exceeds balance"
         );
         unchecked {
             _shares[account] = accountShares - burnShares;
             _totalShares -= burnShares.wadToRay();
         }
         emit Transfer(account, address(0), amount);
         if(block.timestamp > TerminalTimestamp && isTerminal) {
             MTRG.transfer(recipient, amount);
         } else {
             scriptEngine.bucketWithdraw(bucketID, amount, recipient);
         }
     }

    function exit(address recipient) public whenNotPaused {
        address account = msg.sender;
        uint256 accountShares = _shares[account];
        uint256 amount = balanceOf(account);

        unchecked {
            _shares[account] = 0;
            _totalShares -= accountShares.wadToRay();
        }

        emit Transfer(account, address(0), amount);
        if(block.timestamp > TerminalTimestamp && isTerminal) {
             MTRG.transfer(recipient, amount);
        } else {
             scriptEngine.bucketWithdraw(bucketID, amount, recipient);
        }
     }
```

**Fix Comment**

When executeClose() is called, the deposit() and withdraw() are disabled. These two disabled functions can only be re-enabled when admin calls the closeTerminal() to allow users to withdraw their deposited funds from the unbounded (closed) bucket. However, if the admin does not call the closeTerminal(), users may not be able to withdraw their funds. We recommended that the closeTerminal() allows any user to call it and checks if the bucket is in the bucket list since the unbounded (matured) bucket would be removed from the list. This issue has been fixed.

## 5. Asset Loss during Conversion from WstMTRG to stMTRG upon Bucket Closure

| | |
|---|---|
| ID: METER-LSD-05 | Severity: Critical |
| Type: Logic Error | Difficulty: Low |

**Issue**

WstMTRG users may lose their MTRG tokens forever.

If we examine the code below, it becomes evident that the transition from WstMTRG to stMTRG involves referencing the output of the shareToValue function within the stMTRG Contract.

```
// WstMTRG.sol
function unwrap(uint256 _wstMTRGAmount) external returns (uint256) {
        require(_wstMTRGAmount > 0, "wstMTRG: zero amount unwrap not allowed");
        uint256 stMTRGAmount = IStMTRG(stMTRG).shareToValue(_wstMTRGAmount);
        _burn(msg.sender, _wstMTRGAmount);
        IERC20Upgradeable(stMTRG).transfer(msg.sender, stMTRGAmount);
        return stMTRGAmount;
    }
```

https://github.com/meterio/sys-contracts/blob/32b9693224a78a8e11ab4603beb64699ed58b75f/contracts/WstMTRG.sol#L51-L57

```
// stMTRG.sol
function shareToValue(uint256 _share) public view returns (uint256) {
        return _share.wadToRay().rayMul(totalSupply()).rayDiv(_totalShares);
    }
```

https://github.com/meterio/sys-contracts/blob/32b9693224a78a8e11ab4603beb64699ed58b75f/contracts/StMTRG.sol#L194-L196

The shareToValue calls the totalSupply function, which returns 0 if a week has passed since executeClose.

Therefore, if the total supply returns zero, the user calling the unwrap function of the WstMTRG will not be able to receive the stMTRG.

**Recommendation**

Refer to [METER-LSD-04] to change the shareToValue function.

**Fix Comment**

The conversion functions shareToValues and valueToShares use the totalSupply including the MTRG balance of StMTRG. So, they now consider unbounded MTRG tokens after the bucket is closed. However, during this fix review, another issue was found, which was the inconsistency between the totalSupply() function and the _totalSupply value used in the two conversion functions. They fixed the totalSupply() function and the conversion functions use this fixed function for totalSupply.

## 6. Potential Unintended Backdoor Vulnerability in MTRG Token

| | |
|---|---|
| ID: METER-LSD-06 | Severity: Critical |
| Type: Logic Error | Difficulty: Low |

### Issue

Creators of the contract that has MeterERC20 tokens (e.g stMTRG, MTRG, MTR) could rug-pull the tokens from their contracts.

```
function move(
        address _from,
        address _to,
        uint256 _amount
    ) public returns (bool success) {
        require(
            _from == msg.sender ||
                _meterTracker.native_master(_from) == msg.sender,
            "builtin: self or master required"
        );
        _transfer(_from, _to, _amount);
        return true;
    }
```

https://github.com/meterio/sys-contracts/blob/32b9693224a78a8e11ab4603beb64699ed58b75f/contracts/MeterERC20.sol#LL58C1-L70C6

The move function above is similar to the transferFrom function. However, if _from is the address of the msg.sender or the native_master of _from, msg.sender can move the asset at will.

Let's take a look at the code for native_master and see where its value is set.

```
func (s *State) GetMaster(addr meter.Address) meter.Address {
	return meter.BytesToAddress(s.getAccount(addr).Master)
}

// SetMaster set master for the given address.
func (s *State) SetMaster(addr meter.Address, master meter.Address) {
	cpy := s.getAccountCopy(addr)
	if master.IsZero() {
		cpy.Master = nil
	} else {
		cpy.Master = master[:]
	}
	s.updateAccount(addr, &cpy)
}
```

https://github.com/meterio/meter-pov/blob/f3c63e474005266d2ec5d4ac4392b965dc62dd84/state/state.go#L368-L381

```
OnCreateContract: func(_ *vm.EVM, contractAddr, caller common.Address) {
    // set master for created contract
    rt.state.SetMaster(meter.Address(contractAddr), meter.Address(caller))
```

https://github.com/meterio/meter-pov/blob/f3c63e474005266d2ec5d4ac4392b965dc62dd84/runtime/runtime.go#L631-L634

Reviewing the code above, it is clear thatthe address of the creator of the contract is designated as "Master."

This indicates a potential rugpull vector for the accumulated MTRG tokens in stMTRG.

The scope of this concern extends beyond just stMTRG and can potentially impact the entire meter chain ecosystem.

The code should only allow the bucket's owner to modify the candidate, but there is no code to check if the transaction sender is the bucket owner.

Therefore, it was possible to modify the candidate of every existing bucket and steal all the block rewards.

**Recommendation**

Remove the allowance for master from the move functions in MeterERC20.sol and MeterGovERC20.sol, even though they are not in the audit scope.

**Fix Comment**

The move() function has been removed but the patched contract has not been deployed yet. They said they will replace the current contract with the the fixed contract since it requires a hardfork.

## 7. Lack of Blacklist Validation for Deposits and Withdrawals in stMTRG

ID: METER-LSD-07
Type: Logic Error

Severity: High
Difficulty: Low

**Issue**

The blacklisted users can withdraw their funds.

The code below shows that the stMTRG Contract establishes a blacklist and performs sender and receiver validation to prevent malicious users from transferring assets.

```
function _transfer(
        address _from,
        address _to,
        uint256 _value
    ) internal override {
        require(_from != address(0), "ERC20: transfer from the zero address");
        require(_to != address(0), "ERC20: transfer to the zero address");

        _beforeTokenTransfer(_from, _to, _value);
        require(
            !_blackList[_from] && !_blackList[_to],
            "ERC20Pausable: account is in black list"
        );

        uint256 shares = _valueToShare(_value);
        uint256 fromShares = _shares[_from];
        require(fromShares >= shares, "ERC20: transfer amount exceeds balance");
        unchecked {
            _shares[_from] = fromShares - shares;
            _shares[_to] += shares;
        }

        emit Transfer(_from, _to, _value);

        _afterTokenTransfer(_from, _to, _value);
    }
```

https://github.com/meterio/sys-contracts/blob/32b9693224a78a8e11ab4603beb64699ed58b75f/contracts/MeterERC20.sol#L114-L129

Nevertheless, it is worth noting that the withdraw function does not validate whether the withdrawer is in the blacklist.

Consequently, if someone uses the withdraw function to exchange stMTRG tokens for MTRG tokens and subsequently make a deposit using a different address, the blacklist would be useless.

**Recommendation**

Add blacklist functionality to the withdraw functions as well.

**Fix Comment**

The blacklisted user is no longer allowed to withdraw the funds.

## 8. Centralization Risk - 2

ID: METER-LSD-08
Type: Logic Error

Severity: Tips
Difficulty: -

### Issue

Admin can withdraw any user's assets to any address.

```
function adminWithdrawAll(address to) public onlyRole(DEFAULT_ADMIN_ROLE) {
        require(isClosed, "closed!");
        require(
            block.timestamp >= closeTimestamp + CLOSE_DURATION,
            "CLOSE_DURATION!"
        );
        uint256 amount = MTRG.balanceOf(address(this));
        MTRG.transfer(to, amount);
    }

function adminWithdraw(
        address account,
        uint256 amount,
        address recipient
    ) public onlyRole(DEFAULT_ADMIN_ROLE) whenPaused {
        _withdraw(account, amount, recipient);
    }
```

https://github.com/meterio/sys-contracts/blob/32b9693224a78a8e11ab4603beb64699ed58b75f/contracts/StMTRG.sol#L111-L127

The code above is the stMTRG Contract Code. This allows admin to withdraw a user's assets to a specified address. These features are considered potential Rugpull points.

### Recommendation

Remove the adminWithdrawAll function and the adminWithdraw function.

### Fix Comment

The adminWithdrawAll function and the adminWithdraw function have been removed.

## 9. BucketUpdateCandidate does not update the candidate's buckets

|  |  |
|---|---|
| ID: METER-LSD-09 | Severity: High |
| Type: Logic Error | Difficulty: Low |

**Issue**

After the bucket's candidate has been updated to a different candidate, the new candidate's existing buckets/delegators will receive fewer rewards, as the new candidate's total votes (the denominator for calculating shared rewards) will be added.

The following BucketUpdateCandidate code subtracts the bucket's TotalVotes from the original candidate's TotalVotes. It adds it to the new candidate's TotalVotes but does not remove the bucket from the original candidate's bucket list. It does not add the bucket whose candidate is being updated to the new candidate's bucket list.

```go
func (e *MeterTracker) BucketUpdateCandidate(owner meter.Address, id meter.Bytes32,
newCandidateAddr meter.Address) error {
	candidateList := e.state.GetCandidateList()
	bucketList := e.state.GetBucketList()

	b := bucketList.Get(id)
	if b == nil {
		return errBucketNotListed
	}

	nc := candidateList.Get(newCandidateAddr)
	if nc == nil {
		return errCandidateNotListed
	}

	c := candidateList.Get(b.Candidate)
	// subtract totalVotes from old candidate
	if c != nil {
		if c.TotalVotes.Cmp(b.TotalVotes) < 0 {
			return errNotEnoughVotes
		}
		c.TotalVotes.Sub(c.TotalVotes, b.TotalVotes)
	}
	// add totalVotes to new candidate
	nc.TotalVotes.Add(nc.TotalVotes, b.TotalVotes)
	b.Candidate = nc.Addr

	e.state.SetBucketList(bucketList)
	e.state.SetCandidateList(candidateList)
	return nil
}
```

https://github.com/meterio/meter-pov/blob/f3c63e474005266d2ec5d4ac4392b965dc62dd84/builtin/metertracker/meter_tracker_staking.go#L267-L296

```go
func (s *Staking) calcDelegates(env *setypes.ScriptEnv, bucketList *meter.BucketList,
candidateList *meter.CandidateList, inJailList *meter.InJailList) {
	// ...
	for _, c := range candidateList.Candidates {
		delegate := &meter.Delegate{
			Address:     c.Addr,
			PubKey:      c.PubKey,
			Name:        c.Name,
			VotingPower: c.TotalVotes,
			IPAddr:      c.IPAddr,
			Port:        c.Port,
			Commission:  c.Commission,
		}
		// ...
		for _, bucketID := range c.Buckets {
			b := bucketList.Get(bucketID)
			if b == nil {
				s.logger.Info("get bucket from ID failed", "bucketID",
bucketID)
				continue
			}
			// amplify 1e09 because unit is shannon (1e09),  votes of bucket /
votes of candidate * 1e09
			shares := big.NewInt(1e09)
			shares = shares.Mul(b.TotalVotes, shares)
			shares = shares.Div(shares, c.TotalVotes)
			delegate.DistList = append(delegate.DistList,
meter.NewDistributor(b.Owner, b.Autobid, shares.Uint64()))
		}
		delegates = append(delegates, delegate)
	}
	// ...
}
```

https://github.com/meterio/meter-pov/blob/f3c63e474005266d2ec5d4ac4392b965dc62dd84/script/staking/handler_govern.go#L170-L182

## Recommendation

Remove the bucket from the previous candidate's bucket list and add the bucket to the new candidate's bucket list.

## Fix Comment

It removes the bucket from the previous candidate's bucket list and adds the bucket to the new candidate's bucket list.

## 10. Unfair Treatment of MTRG Stakers Due to Price Disparity with MTR

ID: METER-LSD-10
Type: Logic Error

Severity: High
Difficulty: Low

**Issue**

MTRG stakers spend more value than MTR stakers even though they receive the same value of rewards.

```
func (s *Staking) BoundHandler(env *setypes.ScriptEnv, sb *StakingBody, gas uint64)
(leftOverGas uint64, err error) {
...
switch sb.Token {
        case meter.MTR:
                if state.GetEnergy(sb.HolderAddr).Cmp(sb.Amount) < 0 {
                        err = errors.New("not enough meter balance")
                }
        case meter.MTRG:
                if state.GetBalance(sb.HolderAddr).Cmp(sb.Amount) < 0 {
                        err = errors.New("not enough meter-gov balance")
                }
        default:
                err = errInvalidToken
        }
...
bucket := meter.NewBucket(sb.HolderAddr, candAddr, sb.Amount, uint8(sb.Token), opt, rate,
sb.Autobid, ts, nonce)
...
}
```

https://github.com/meterio/meter-pov/blob/f3c63e474005266d2ec5d4ac4392b965dc62dd84/script/staking/handler_bound.go#L10

There is a significant price difference between MTR (Meter) and MTRG (Meter Governance) tokens.

However, the system does not account for this price disparity when calculating staking amounts.

Instead, it treats equal staking amounts of both tokens as having the same value and consequently assigns equal rewards.

This practice creates an unfair situation for all MTRG stakers, including those involved in the stMTRG contract.

The absence of consideration for the price difference leads to an unfair outcome.

This practice undermines fairness within the staking process and need to warn to users.

**Recommendation**

We recommend disabling MTR Staking.

**Fix Comment**

MTR Staking has been disabled.

## DISCLAIMER

This report does not guarantee investment advice, the suitability of the business models, and codes that are secure without bugs. This report shall only be used to discuss known technical issues. Other than the issues described in this report, undiscovered issues may exist such as defects on the main network. In order to write secure codes, correction of discovered problems and sufficient testing thereof are required.

# Appendix. A

## Severity Level

| | |
|---|---|
| **CRITICAL** | Must be addressed as a vulnerability that has the potential to seize or freeze substantial sums of money. |
| **HIGH** | Has to be fixed since it has the potential to deny users compensation or momentarily freeze assets. |
| **MEDIUM** | Vulnerabilities that could halt services, such as DoS and Out-of-Gas, need to be addressed. |
| **LOW** | Issues that do not comply with standards or return incorrect values |
| **TIPS** | Tips that makes the code more usable or efficient when modified |

## Difficulty Level

| | **Low** | **Medium** | **High** |
|---|---|---|---|
| **Privilege** | anyone | Miner/Block Proposer | Admin/Owner |
| **Capital needed** | Small or none | Gas fee or volatile as price change | More than exploited amount |
| **Probability** | 100% | Depend on environment | Hard as mining difficulty |

## Vulnerability Category

| | |
|---|---|
| **Arithmetic** | • Integer under/overflow vulnerability<br>• floating point and rounding accuracy |
| **Access & Privilege Control** | • Manager functions for emergency handle<br>• Crucial function and data access<br>• Count of calling important task, contract state change, intentional task delay |
| **Denial of Service** | • Unexpected revert handling<br>• Gas limit excess due to unpredictable implementation |
| **Miner Manipulation** | • Dependency on the block number or timestamp.<br>• Frontrunning |
| **Reentrancy** | •Proper use of Check-Effect-Interact pattern.<br>•Prevention of state change after external call<br>• Error handling and logging. |
| **Low-level Call** | • Code injection using delegatecall<br>• Inappropriate use of assembly code |
| **Off-standard** | • Deviate from standards that can be an obstacle of interoperability. |
| **Input Validation** | • Lack of validation on inputs. |
| **Logic Error/Bug** | • Unintended execution leads to error. |
| **Documentation** | •Coherency between the documented spec and implementation |
| **Visibility** | • Variable and function visibility setting |
| **Incorrect Interface** | • Contract interface is properly implemented on code. |

End of Document