SECURITY AUDIT REPORT FOR EVEREST GOLD SMART CONTRACTS

PUBLIC REPORT

JUNE 01, 2020

✓erichains Lab

info@verichains.io

https://www.verichains.io

Driving Technology >> Forward

EXECUTIVE SUMMARY

This Security Audit Report prepared by Verichains Lab on May 21, 2020. We would like to thank Everest Gold to trust Verichains Lab to audit smart contracts. Delivering high-quality audits is always our top priority.

This audit focused on identifying security flaws in code and the design of the smart contracts. The scope of the audit is limited to the source code files provided to Verichains Lab on May 18, 2020. Verichains Lab completed the assessment using manual, static, and dynamic analysis techniques.

The assessment identified some issues in Everest Gold smart contracts code. Overall, the code reviewed is of good quality, written with the awareness of smart contract development best practices.

CONTENTS

ACRONYMS AND ABBREVIATIONS

Ethereum	An open source platform based on blockchain technology to create and distribute smart contracts and decentralized applications.
ETH (Ether)	A cryptocurrency whose blockchain is generated by the Ethereum platform. Ether is used for payment of transactions and computing services in the Ethereum network.
Smart contract	A computer protocol intended to digitally facilitate, verify or enforce the negotiation or performance of a contract.
Solidity	A contract-oriented, high-level language for implementing smart contracts for the Ethereum platform.
Solc	A compiler for Solidity.
EVM	Ethereum Virtual Machine.

AUDIT OVERVIEW

ABOUT EVEREST GOLD

> *Everest Gold is a regulated dealer of precious metals in Singapore, subject to the PSPM Act 2019 (Precious Stones and Precious Metals Act, License no. PS20190001500, Issued by Ministry of Law).*
>
> *Everest Gold utilizes encrypted technology and world-class infrastructure to provide a secured trading platform for gold investors. Our trading platform is created to solve various issues commonly faced by gold investors. (Risk, storage, transaction, ownership, etc.) The 24/7, advanced system allows users to grasp every price fluctuation to seize the best trading opportunities, execute instant transaction and circulation to optimise profits in every trade.*
>
> *Combining finance with cutting-edge technology, we're committed to bringing you the most efficient and secured trading platform in the world!*

Everest Gold is a next-generation trading platform for gold investors. Based on blockchain technology, they are running a promising platform that inherits all features that help traders confidently put their assets in to search for profits in open, secure and auditable environment.

SCOPE OF THE AUDIT

This audit focused on identifying security flaws in code and the design of the smart contracts. It was conducted on commit `7e87af3f81ac0b44aad2ab93805b471028d1b9a9` sent to VeriChains on May 18, 2020.

The final report is based on the update patches on commit `feb442ab545c7d0787d6ddbc46d566ae740f4f17` by Everest Gold on May 25, 2020.

AUDIT METHODOLOGY

Our security audit process for smart contract includes two steps:

- Smart contract codes are scanned/tested for commonly known and more specific vulnerabilities using public and RK87, our in-house smart contract security analysis tool.
- Manual audit of the codes for security issues. The contracts are manually analyzed to look for any potential problems.

Following is the list of commonly known vulnerabilities that was considered during the audit of the smart contract:

- Integer Overflow and Underflow
- TimeStamp Dependence
- Race Conditions
- Transaction-Ordering Dependence
- DoS with (Unexpected) revert
- DoS with Block Gas Limit
- Gas Usage, Gas Limit and Loops
- Redundant fallback function
- Unsafe type Inference
- Reentrancy
- Explicit visibility of functions state variables (external, internal, private and public)
- Logic Flaws

For vulnerabilities, we categorize the findings into categories, depending on their criticality:

LOW	An issue that does not have a significant impact, can be considered as less important
MEDIUM	A vulnerability that could affect the desired outcome of executing the contract with medium impact in a specific scenario; needs to be fixed.
HIGH	A vulnerability that could affect the desired outcome of executing the contract with high impact; needs to be fixed with high priority.
CRITICAL	A vulnerability that can disrupt the contract functioning; creates a critical risk to the contract; required to be fixed immediately.

AUDIT RESULTS

Everest Gold implemented various smart contracts with very good logic separation and optimization. The contracts are coded with deep understanding on how EVM and Solidity works. Testcases are fully prepared for all the contracts. The main contracts of Everest Gold are:

- `BlackListController`, `WhitelistController` and `SecretSuperAdmin` controllers that implemented in multi-sigs approach. These controller are used in `EGToken` contract to enhance ERC20 in a way that token owners can be property controlled for a safer trading environment.
- `MetaEGTProducer` is a multi-sigs contract that can mint EGT token.
- `MultiSigWallet` is a multi-sigs wallet contract.
- `EGToken` that is based on ERC20 and implementation include features of ERC223.
- `MetaEGNetwork` is the network to mint EG tokens.

The smart contracts are highly optimized with inline assembly that make our static verification a bit more difficult.

In summary all the features are implemented very carefully. We only identified one security bug, some typing errors and provide some suggestions for gas optimization in the following section.

Update:

- May 25, 2020. Everest Gold fixed all the found threats and other related sugesstions.

VULNERABILITIES FINDINGS

FIXED **HIGH**
REENTRANCY IN MULTIOWNER.EXECUTETRANSACTION

```
function executeTransaction(uint transactionId, bytes32 transactionData)
    public
    ownerExist(msg.sender)
    confirmed(transactionId, msg.sender)
    transactionExists(transactionId, transactionData)
    notExecuted(transactionId)
{
    if (isConfirmed(transactionId)) {
        Transaction storage txn = _transactions[transactionId];
        require(txn._transactionData == transactionData);
        if ( external_call(txn._destination, txn._value, txn._data.length, txn._data)) {
            emit Execution(transactionId, transactionData);
            txn._executed = true;
        } else {
            emit ExecutionFailure(transactionId, transactionData);
        }
    }
}
```

Execution flag `txn._executed` is set after `external_call` results in reentrancy bug. Other similar functions already applied the fix.

RECOMMENDED FIXES
Change the code as below:

```
    if (isConfirmed(transactionId)) {
        Transaction storage txn = _transactions[transactionId];
        require(txn._transactionData == transactionData);
        txn._executed = true;
        if ( external_call(txn._destination, txn._value, txn._data.length, txn._data)) {
            emit Execution(transactionId, transactionData);
        } else {
            txn._executed = false;
            emit ExecutionFailure(transactionId, transactionData);
        }
    }
```

May 25, 2020. Everest Gold fixed this in branch `master` in commit *`feb442ab545c7d0787d6ddbc46d566ae740f4f17`*.

FIXED **LOW**

GAS SAVING

METAEGNETWORK.CREATEGOLDTOKENISATION

```
    /**
    *@dev created a new EGT Producer in the case all parties haven't  created an EGT producer
    */
    function createGoldTokenisation(address sender, address[] memory keyList, uint8[] memory
indexRoles, uint8 required,  address mintedWallet)
    public
    onlyEverestAdmin
    {
        require(keyList.length == indexRoles.length);
        require(required <= keyList.length);
        require(required >=2);
        bytes memory indexInput;
        for(uint i = 0; i < keyList.length;i++) {
           require(_everestRoles.isValidIndexAccount(indexRoles[i], keyList[i]));
            indexInput = abi.encodePacked(indexInput,keyList[i]);
        }

        bytes32 index = keccak256(indexInput);

        if( registeredInstantiations[index] == address(0)){
            registeredInstantiations[index] = _createGoldProducer(sender, keyList, required,
mintedWallet);
        }
    }
```

The final `if` should be changed to `require` so that remain gas can be refunded if the transaction is submitted and supported client can alert the user if the transaction is about to submit:

```
        require(registeredInstantiations[index] == address(0));
```

May 25, 2020. Everest Gold used this optimization in branch `master` in commit *`feb442ab545c7d0787d6ddbc46d566ae740f4f17`*.

METAEGTPRODUCER.ISCONFIRMED

```
    /// @dev Returns the confirmation status of a transaction.
    /// @param transactionId Transaction ID.
    /// @return Confirmation status.
    function isConfirmed(uint transactionId)
    public
    view
    returns (bool)
    {
        uint8 count = 0;
        for (uint i=0; i< _tokenizers.length; i++) {
            if (_confirmations[transactionId][_tokenizers[i]])
                count += 1;
            if (count == _required)
                return true;
        }
    }
```

Move the second `if` so that it only check after counter variable is changed, and also compare with constant and use `uint` for counter variable to reduce some gas cost.

```
        uint need = _required;
        if (need == 0) return true; // if _required is always > 0, we can remove this line
        for (uint i=_tokenizers.length - 1; i!=uint(-1); i--) {
            if (_confirmations[transactionId][_tokenizers[i]]) {
                need -= 1;
                if (need == 0) return true; // we can swap this line with previous and change
constant to 1, but that will make it looks ugly
            }
        }
```

Almost all other for loops can be optimized using this method too (`getConfirmationCount`, `getTransactionCount`,…).

May 25, 2020. Everest Gold used this optimization in branch *`master`* in commit *`feb442ab545c7d0787d6ddbc46d566ae740f4f17`*.

METAEGTPRODUCER.GETCONFIRMATIONS, METAEGTPRODUCER.GETTRANSACTIONIDS

```
function getConfirmations(uint transactionId)
public
view
returns (address[] memory confirmations)
{
    address[] memory confirmationsTemp = new address[](_tokenizers.length);
    uint count = 0;
    uint i;
    for (i=0; i<_tokenizers.length; i++)
        if (_confirmations[transactionId][_tokenizers[i]]) {
            confirmationsTemp[count] = _tokenizers[i];
            count += 1;
        }
    confirmations = new address[](count);
    for (i=0; i<count; i++)
        confirmations[i] = confirmationsTemp[i];
}
```

We don't need to allocate 2 arrays for this method, as solidity doesn't really have allocation/deallocation concepts like normal systems. We just need to create new length for the array afterward:

```
function getConfirmations(uint transactionId)
public
view
returns (address[] memory confirmations)
{
    confirmations = new address[](_tokenizers.length);
    uint count = 0;
    // if reversing the result is okey, we can use the reversing loop to reduce gas cost
    for (uint i=0; i<_tokenizers.length; i++)
        if (_confirmations[transactionId][_tokenizers[i]]) {
            confirmations[count] = _tokenizers[i];
            count += 1;
        }
    assembly {
        mstore(confirmations, count) // store the length
    }
}
```

Another optimization method for array is shifting the base address, in the `getTransactionIds` method:

```
    function getTransactionIds(uint from, uint to, bool pending, bool executed)
    public
    view
    returns (uint[] memory _transactionIds)
    {
        uint[] memory transactionIdsTemp = new uint[](_transactionCount);
        uint count = 0;
        uint i;
        for (i=0; i<_transactionCount; i++)
            if (   pending && !_transactions[i].executed
            || executed && _transactions[i].executed)
            {
                transactionIdsTemp[count] = i;
                count += 1;
            }
        _transactionIds = new uint[](to - from);
        for (i=from; i<to; i++)
            _transactionIds[i - from] = transactionIdsTemp[i];
    }
```

We can optimize gas by storing `transactionIds` array with the shifted base of the `transactionIdsTemp` array:

```
        uint[] memory transactionIdsTemp = new uint[](_transactionCount);
        uint count = 0;
        uint i;
        for (i=0; i<_transactionCount; i++)
            if (   pending && !_transactions[i].executed
            || executed && _transactions[i].executed)
            {
                transactionIdsTemp[count] = i;
                count += 1;
                if(count >= to) break;
            }
        assembly {
            _transactionIds:=add(transactionIdsTemp,mul(from,32)) // "point" _transactionIds
to the new offset of transactionIdsTemp, 32 is sizeof uint
            mstore(_transactionIds,sub(to,from)) // store the length
        }
```

May 25, 2020. Everest Gold used this optimization in branch `master` in commit `feb442ab545c7d0787d6ddbc46d566ae740f4f17`.

MULTIOWNER.GETTRANSACTIONCOUNT

```
    function getTransactionCount(bool pending, bool executed)
        public
        view
        returns ( uint count)
    {
        for ( uint i = 0; i < _transactionCount; i++) {
            if(pending && !_transactions[i]._executed
                || executed && _transactions[i]._executed)
            count += 1;
        }
    }
```

Gas cost of the function can be reduced by handling each case of `pending` and `executed` separately:

```
    function getTransactionCount(bool pending, bool executed)
        public
        view
        returns ( uint count)
    {
        if (!pending && !executed) return 0;
        if (pending && executed) return _transactionCount;
        for ( uint i = 0; i < _transactionCount; i++) {
            if(_transactions[i]._executed)
                count += 1;
        }
        if (pending) count = _transactionCount - count;
    }
```

But the optimized version reduced code readability, so the optimized code is here just for reference. The same approach can be used for `getTransactionIds` too.

FIXED **LOW**

TYPOS

- `WhitelistController` contract is in wrong folder name: `WhileList`.
- `MetaEGTProducer.sol`:
    - `/// @dev Returns total number of transactions after filers are applied.`

May 25, 2020. Everest Gold fixed these and also related ones in branch *master* in commit *feb442ab545c7d0787d6ddbc46d566ae740f4f17*.

CONCLUSION

Everest Gold smart contracts have been audited by Verichains Lab using various public and in-house analysis tools and intensively manual code review. The assessment identified some issues and provided some suggestions in Everest Gold smart contracts code. After the first report, Everest Gold fixed all the threats and related suggestions. Overall, the code reviewed is of good quality, written with the awareness of smart contract development best practices.

LIMITATIONS

Please note that security auditing cannot uncover all existing vulnerabilities, and even an audit in which no vulnerabilities are found is not a guarantee for a 100% secure smart contract. However, auditing allows discovering vulnerabilities that were unobserved, overlooked during development and areas where additional security measures are necessary.

APPENDIX I

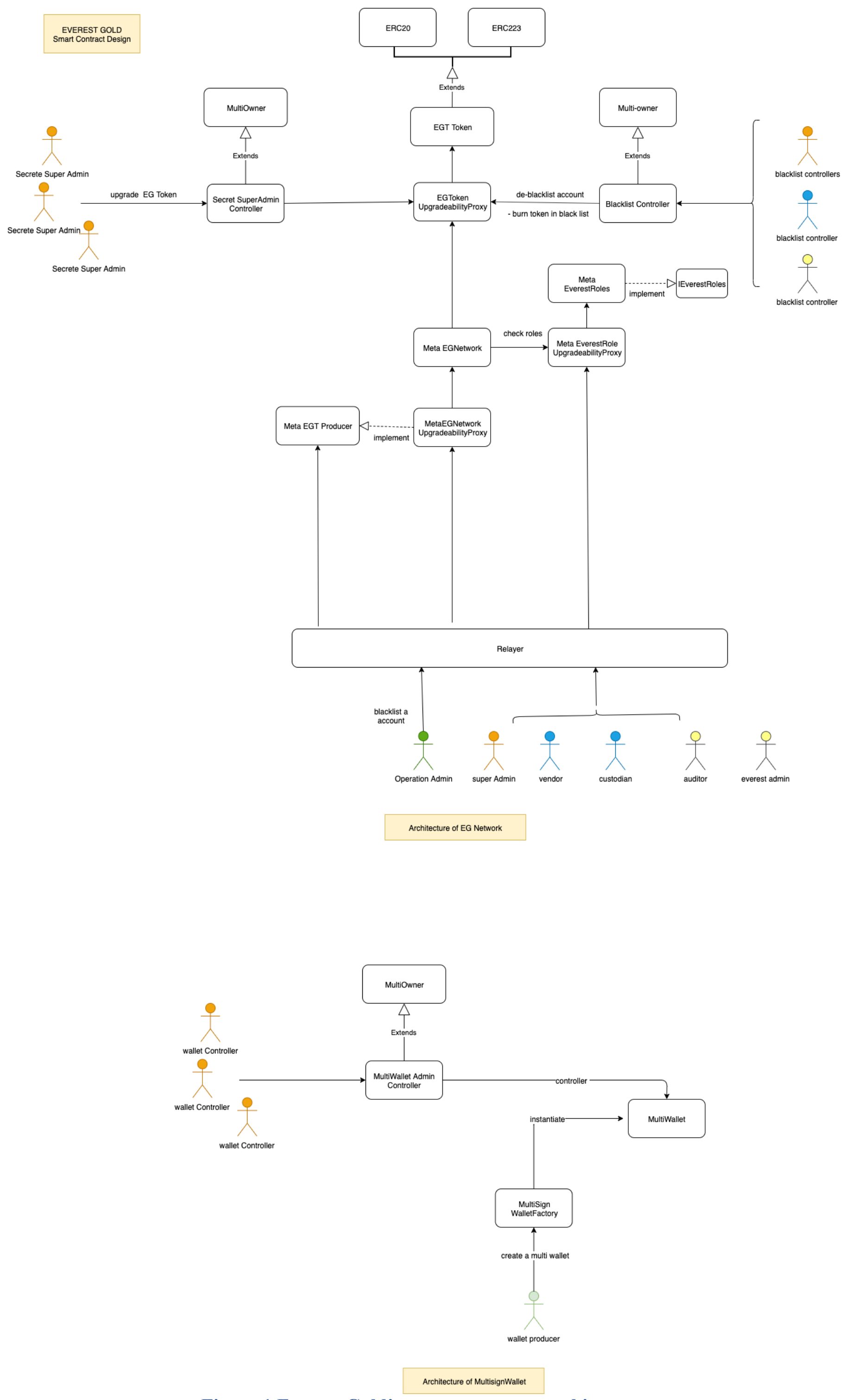

Figure 1 Everest Gold's smart contracts architecture

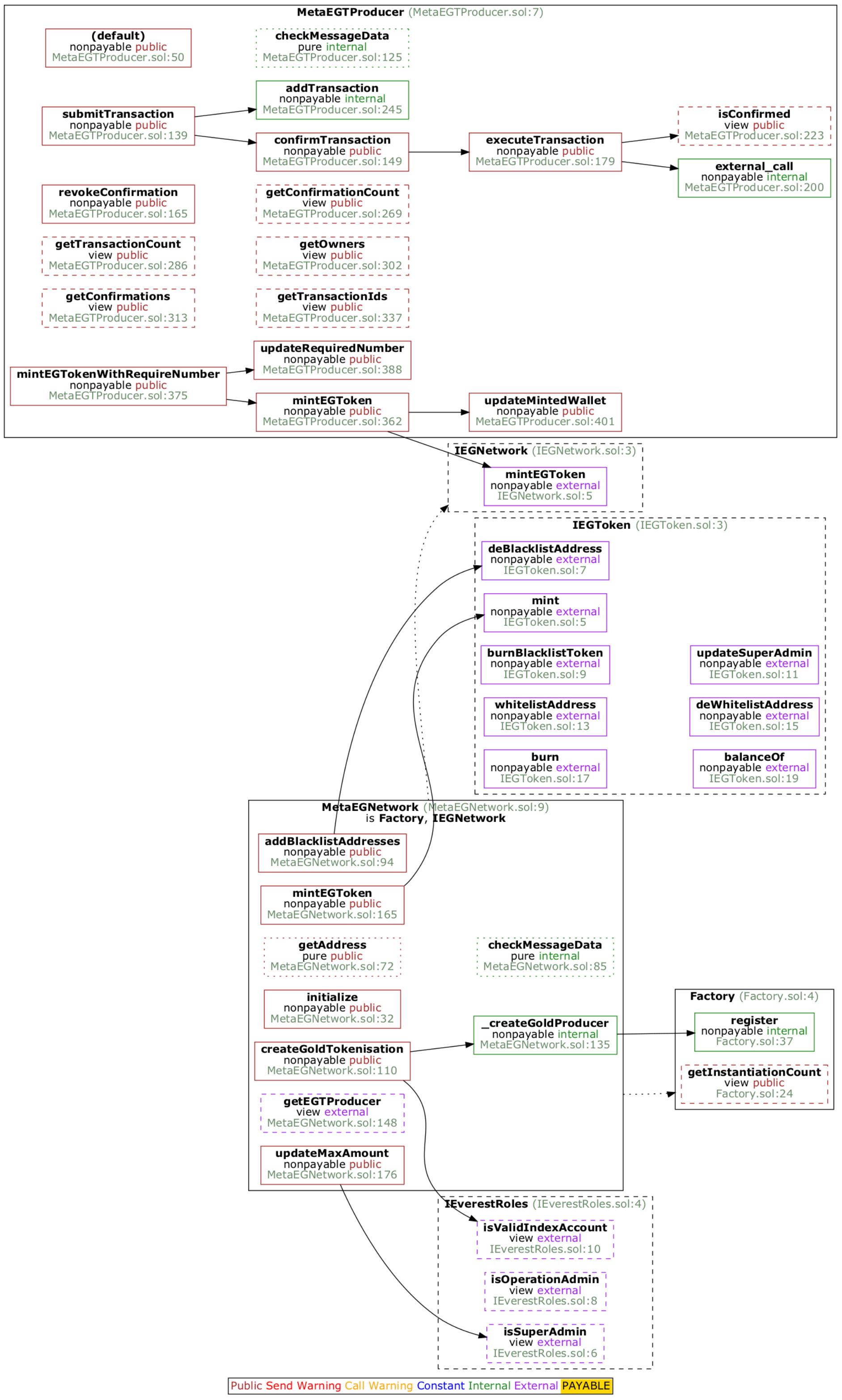
MetaEGTProducer (MetaEGTProducer.sol:7)
(default) nonpayable public MetaEGTProducer.sol:50
checkMessageData pure internal MetaEGTProducer.sol:125
addTransaction nonpayable internal MetaEGTProducer.sol:245
submitTransaction nonpayable public MetaEGTProducer.sol:139
confirmTransaction nonpayable public MetaEGTProducer.sol:149
executeTransaction nonpayable public MetaEGTProducer.sol:179
isConfirmed view public MetaEGTProducer.sol:223
external_call nonpayable internal MetaEGTProducer.sol:200
revokeConfirmation nonpayable public MetaEGTProducer.sol:165
getConfirmationCount view public MetaEGTProducer.sol:269
getTransactionCount view public MetaEGTProducer.sol:286
getOwners view public MetaEGTProducer.sol:302
getConfirmations view public MetaEGTProducer.sol:313
getTransactionIds view public MetaEGTProducer.sol:337
updateRequiredNumber nonpayable public MetaEGTProducer.sol:388
mintEGTokenWithRequireNumber nonpayable public MetaEGTProducer.sol:375
mintEGToken nonpayable public MetaEGTProducer.sol:362
updateMintedWallet nonpayable public MetaEGTProducer.sol:401
IEGNetwork (IEGNetwork.sol:3)
mintEGToken nonpayable external IEGNetwork.sol:5
IEGToken (IEGToken.sol:3)
deBlacklistAddress nonpayable external IEGToken.sol:7
mint nonpayable external IEGToken.sol:5
burnBlacklistToken nonpayable external IEGToken.sol:9
updateSuperAdmin nonpayable external IEGToken.sol:11
whitelistAddress nonpayable external IEGToken.sol:13
deWhitelistAddress nonpayable external IEGToken.sol:15
burn nonpayable external IEGToken.sol:17
balanceOf nonpayable external IEGToken.sol:19
MetaEGNetwork (MetaEGNetwork.sol:9) is Factory, IEGNetwork
addBlacklistAddresses nonpayable public MetaEGNetwork.sol:94
mintEGToken nonpayable public MetaEGNetwork.sol:165
getAddress pure public MetaEGNetwork.sol:72
checkMessageData pure internal MetaEGNetwork.sol:85
initialize nonpayable public MetaEGNetwork.sol:32
_createGoldProducer nonpayable internal MetaEGNetwork.sol:135
createGoldTokenisation nonpayable public MetaEGNetwork.sol:110
getEGTProducer view external MetaEGNetwork.sol:148
updateMaxAmount nonpayable public MetaEGNetwork.sol:176
Factory (Factory.sol:4)
register nonpayable internal Factory.sol:37
getInstantiationCount view public Factory.sol:24
IEverestRoles (IEverestRoles.sol:4)
isValidIndexAccount view external IEverestRoles.sol:10
isOperationAdmin view external IEverestRoles.sol:8
isSuperAdmin view external IEverestRoles.sol:6
Public Send Warning Call Warning Constant Internal External PAYABLE

Figure 2 Call-graph of MetaEGNetwork

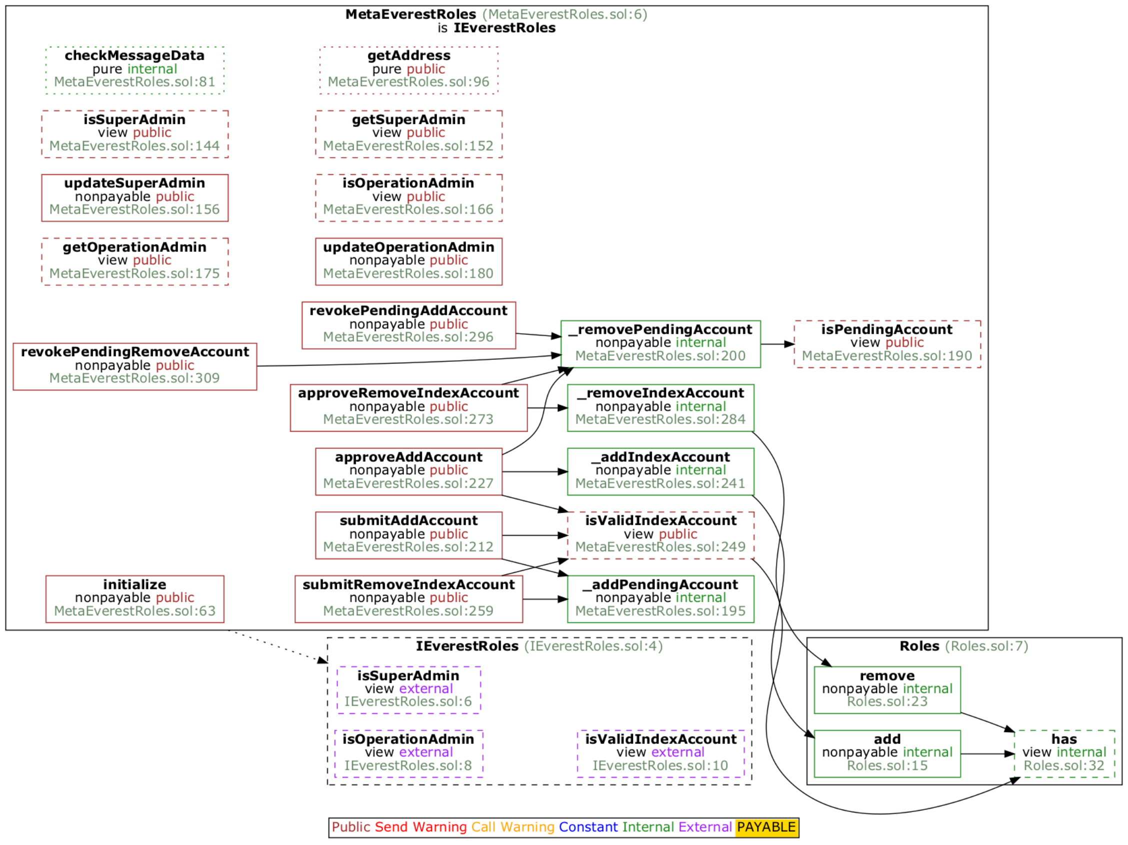

Figure 3 Call-graph of MetaEverestRolesMock

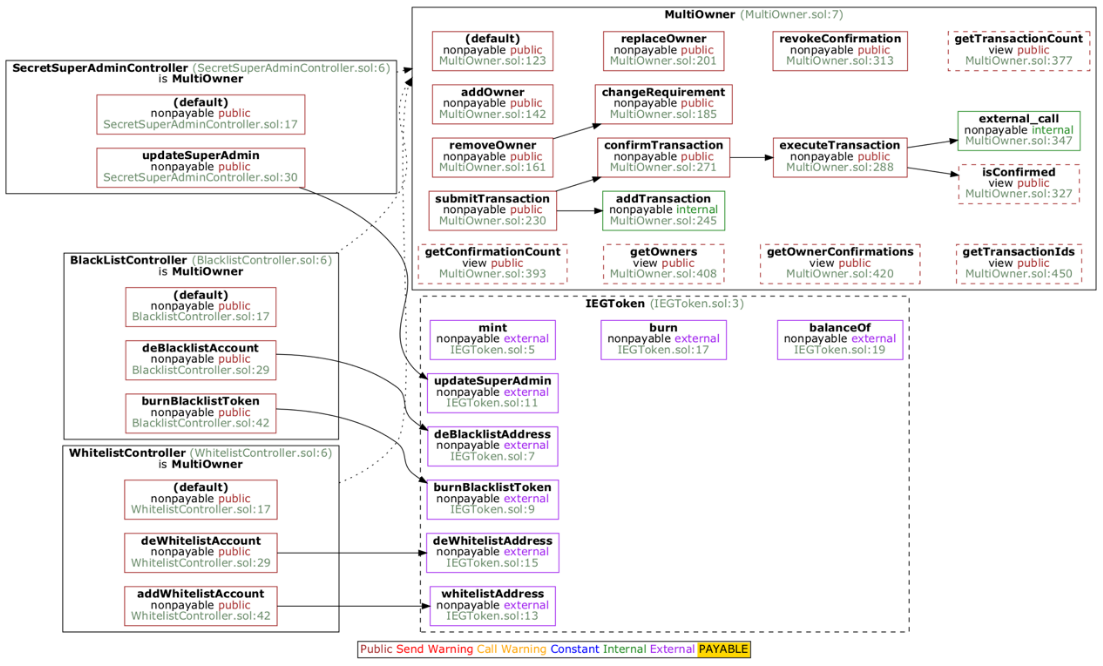

Figure 4 Call-graph of controllers contracts

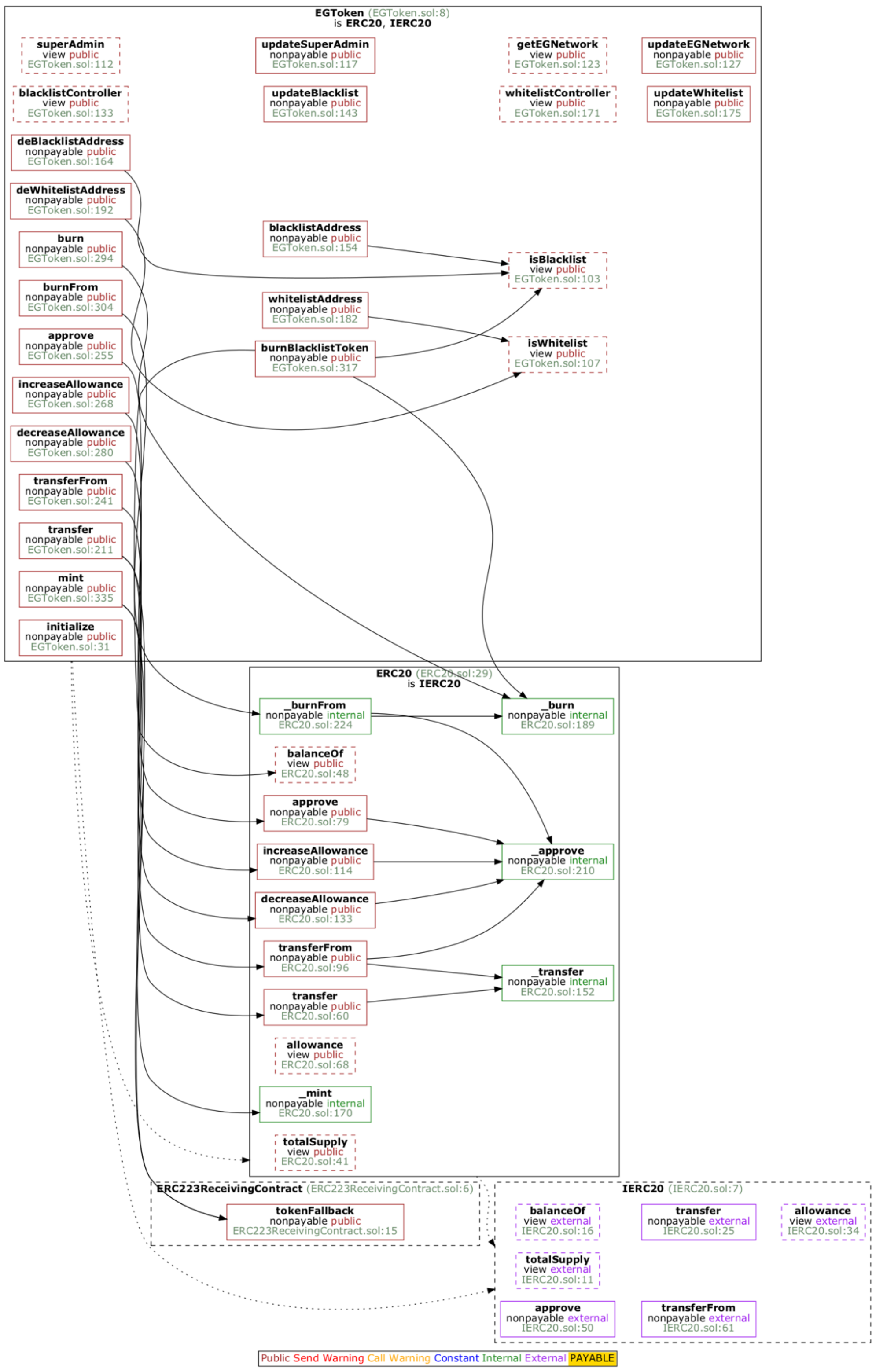
EGToken (EGToken.sol:8)
is ERC20, IERC20
superAdmin view public EGToken.sol:112
updateSuperAdmin nonpayable public EGToken.sol:117
getEGNetwork view public EGToken.sol:123
updateEGNetwork nonpayable public EGToken.sol:127
blacklistController view public EGToken.sol:133
updateBlacklist nonpayable public EGToken.sol:143
whitelistController view public EGToken.sol:171
updateWhitelist nonpayable public EGToken.sol:175
deBlacklistAddress nonpayable public EGToken.sol:164
deWhitelistAddress nonpayable public EGToken.sol:192
blacklistAddress nonpayable public EGToken.sol:154
isBlacklist view public EGToken.sol:103
burn nonpayable public EGToken.sol:294
burnFrom nonpayable public EGToken.sol:304
whitelistAddress nonpayable public EGToken.sol:182
isWhitelist view public EGToken.sol:107
approve nonpayable public EGToken.sol:255
burnBlacklistToken nonpayable public EGToken.sol:317
increaseAllowance nonpayable public EGToken.sol:268
decreaseAllowance nonpayable public EGToken.sol:280
transferFrom nonpayable public EGToken.sol:241
transfer nonpayable public EGToken.sol:211
mint nonpayable public EGToken.sol:335
initialize nonpayable public EGToken.sol:31
ERC20 (ERC20.sol:29)
is IERC20
_burnFrom nonpayable internal ERC20.sol:224
_burn nonpayable internal ERC20.sol:189
balanceOf view public ERC20.sol:48
approve nonpayable public ERC20.sol:79
_approve nonpayable internal ERC20.sol:210
increaseAllowance nonpayable public ERC20.sol:114
decreaseAllowance nonpayable public ERC20.sol:133
transferFrom nonpayable public ERC20.sol:96
_transfer nonpayable internal ERC20.sol:152
transfer nonpayable public ERC20.sol:60
allowance view public ERC20.sol:68
_mint nonpayable internal ERC20.sol:170
totalSupply view public ERC20.sol:41
ERC223ReceivingContract (ERC223ReceivingContract.sol:6)
tokenFallback nonpayable public ERC223ReceivingContract.sol:15
IERC20 (IERC20.sol:7)
balanceOf view external IERC20.sol:16
transfer nonpayable external IERC20.sol:25
allowance view external IERC20.sol:34
totalSupply view external IERC20.sol:11
approve nonpayable external IERC20.sol:50
transferFrom nonpayable external IERC20.sol:61
Public Send Warning Call Warning Constant Internal External PAYABLE

Figure 5 Call-graph of EGToken

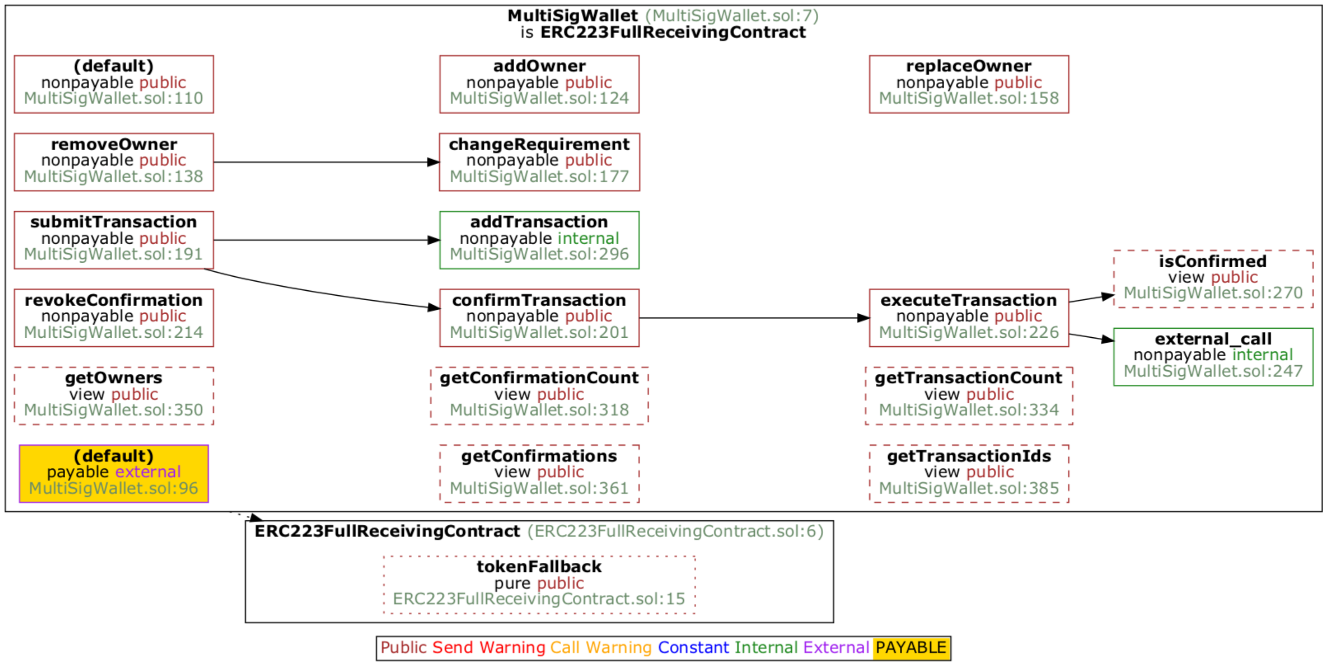

Figure 6 Call-graph of MultiSigWallet