USDKG

Date	January 2025

1 Executive Summary

This report presents the results of our engagement with **Karat USD** to review **USDKG**.

The review was conducted over one week, from **January 20, 2025** to **January 24, 2025**, by **George Kobakhidze** and **Rai Yang**. A total of 10 person-days were spent.

USDKG is a gold-backed stablecoin represented as ERC20 token. The token's constructor uses an owner and a compliance address for administrative functions. Both owner and compliance address are meant to be governed by a Multisig contract based on Gnosis Safe multisig contract. Certain functions are removed such as upgradability, offchain approval with signatures and abilities to modify owners and threshold. The Karat team should note and follow updates to the Safe (previously Gnosis Safe) multisig contracts in order to keep the USDKG multisig contracts up to date with latest ecosystem developments and any possible issues.

2 Scope

Our review focused on the commit hash 5e2b92515976fa331291c35e78ab6cb13542b30d. The list of files in scope can be found in the Appendix.

2.1 Objectives

Together with the **Karat USD** team, we identified the following priorities for our review:

1. Correctness of the implementation, consistent with the intended functionality and without unintended edge cases.
2. Identify known vulnerabilities particular to smart contract systems, as outlined in our Smart Contract Best Practices, and the Smart Contract Weakness Classification Registry.

3 Security Specification

This section describes, **from a security perspective**, the expected behavior of the system under audit. It is not a substitute for documentation. The purpose of this section is to identify specific security properties that were validated by the audit team.

3.1 Actors

The relevant actors are listed below with their respective abilities:

- USDKG deployers. The deployers create the contracts and are able to set the relevant constructor parameters, such as owner and compliance addresses for the USDKG token.
- USDKG owners. They are able to pause the contract, manage token supply via `issue` and `redeem`, and set transfer fee parameters.
- USDKG compliance team. The compliance team is responsible for managing the token blacklist and burning of fraudulent tokens.
- Multisig owners. The multisig owners are responsible for signing off on transactions to be approved and subsequently executed.
- Token users. The token users may utilize the token as they see fit, such as transfer.

3.2 Trust Model

In any system, it's important to identify what trust is expected/required between various actors. For this audit, we established the following trust model:

- USDKG deployers. The deployers need to set the right token and multisig parameters in the constructor to ensure system stability. For example, a threshold that is too low on a multisig may compromise the multisig's security.
- USDKG owners. The token owners have almost complete control over the token and its supply. Via the `issue` and `redeem` functions, they can take away anyone's tokens and create them for themselves. A compromised owner set may cause critical damage to the system, so it needs to be secured with a sufficiently large multisig owner set.
- USDKG compliance team. Similarly to the owners, the compliance team has significant control over the funds. Namely, they may stop anyone they choose from owning and interacting with the USDKG token and may even burn any one user's USDKG balance by adding them to the blacklist and destroying their funds via `destroyBlackFunds`. The USDKG compliance team also must be behind a sufficiently large multisig set.
- Multisig owners. As the ultimate authority on what the multisig actually transacts, the multisig owners need to be vigilant and careful with their signatures so they don't compromise any systems dependent on the multisig. The higher the threshold and owner count in the multisig, the less trust each multisig owner is given.
- Token users. The token users are not trusted in the system in any manner.

4 Findings

Each issue has an assigned severity:

- Minor issues are subjective in nature. They are typically suggestions around best practices or readability. Code maintainers should use their own judgment as to whether to address such issues.
- Medium issues are objective in nature but are not security vulnerabilities. These should be addressed unless there is a clear reason not to.
- Major issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- Critical issues are directly exploitable security vulnerabilities that need to be fixed.

4.1 `transferFrom()` Lacks `notBlackListed` Modifier on the Spender `msg.sender` Medium ✓ Fixed

Resolution

Fixed in commit 0d22c5326e21541df0c718db98004d5a475aa2ea by putting the `notBlackListed` modifier on `msg.sender` in the `transferFrom()` function as well.

Description

The USDKG token has functionality to blacklist users from using it. For example, a `notBlackListed` modifier exists to verify that a user does not belong to a blacklisted list:

contracts/USDKG.sol:L86-L92

```
/**
 * @dev Modifier to make a function callable only when sender is not blacklisted.
 */
modifier notBlackListed(address sender) {
    require(!isBlackListed[sender], "user blacklisted");
    _;
}
```

This is present on functions `transfer()` and `transferFrom()` where it is checking the `msg.sender` and `_from` addresses respectively:

contracts/USDKG.sol:L103

```
function transfer(address _to, uint256 _value) public whenNotPaused notBlackListed(msg.sender) returns (bool) {
```

contracts/USDKG.sol:L122

```
function transferFrom(address _from, address _to, uint256 _value) public whenNotPaused notBlackListed(_from) returns (bool) {
```

However, in the case of `transferFrom()` it would also be valuable to check blacklisting against the spender, i.e. `msg.sender` as well. That is because a malicious or a compromised spender who received approval from a victim may be identified as an attacker prior to them executing, or perhaps continuing the execution of, exploits. For example, one such compromised spender may be a vulnerable smart contract that has the USDKG token as part of its system, like collateral or a lending asset. An attacker may execute an exploit on such a contract that has approval on it from its victims. The exploit could move the tokens from the victims (the `_from` address) to the vulnerable smart contract (the `msg.sender`) via `transferFrom()`, perform the exploit, and then move the tokens to the attacker via `transfer()`. Blacklisting the attacker themselves wouldn't be useful as they can simply spin up another account and activate the exploit from there. However, blacklisting the vulnerable smart contract itself would prevent the `transferFrom()` operation from the victim to the contract, thereby stopping the exploit at least as far as the USDKG token is concerned. Moreover, this could be used by the compliance team of the USDKG token to blacklist known phishing contracts or otherwise potentially sanctioned contracts as deemed appropriate by the USDKG team.

Of course, this is contingent on the USDKG compliance team knowing the vulnerable or inappropriate contracts in the first place prior to the malicious activity taking place. This is possible today with monitoring solutions as well.

Recommendation

Apply the `notBlacklisted` modifier to the spender in the `transferFrom()` function, i.e. `msg.sender` as well.

4.2 Missing Validation for Parameters in the Constructor Medium ✓ Fixed

Resolution

Fixed in commit 0d22c5326e21541df0c718db98004d5a475aa2ea by introducing 0-checks on the `owner` and `compliance` addresses in the constructor.

Description

The constructor function of the `USDKG` contract does not validate that the `owner` and `compliance` address parameters are non-zero. If either address is set to the zero address, the token contract would become unusable.

Examples

contracts/USDKG.sol:L45-L52

```
constructor (address _owner, address _compliance) {
    owner = _owner;
    compliance = _compliance;
    _totalSupply = 0;
    name = "USDKG";
    symbol = "USDKG";
    decimals = 6;
}
```

Recommendation

Add non-zero address validation in the constructor.

4.3 Small Amount Can Ignore Fee on Transfer Minor Acknowledged

Resolution

Acknowledged and accepted the risk by the client with the following note:

> We currently have no plans to enable a fee on transfer and it's acceptable for us to ignore very small transfers (fractions of a cent)

Description

The USDKG token has admin functionality to enable fees on transfer as represented by the `basisPointsRate` variable:

contracts/USDKG.sol:L103-L104

```
function transfer(address _to, uint256 _value) public whenNotPaused notBlackListed(msg.sender) returns (bool) {
    uint256 fee = _value * basisPointsRate / FEE_PRECISION;
```

contracts/USDKG.sol:L122-L128

```
function transferFrom(address _from, address _to, uint256 _value) public whenNotPaused notBlackListed(_from) returns (bool) {
    uint256 _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;
```

However, since `fee` is a result of division by `FEE_PRECISION` in both cases, it is vulnerable to being rounded down to zero if the numerator is less than `FEE_PRECISION`. `FEE_PRECISION` is a constant of `10000`:

contracts/USDKG.sol:L22

```
uint256 public constant FEE_PRECISION = 10000;
```

So, a user seeking to ignore the fee may construct transfers with sufficiently small amounts such that `_value * basisPointsRate` is less than `10000`, making the resulting `fee` to be `0`. This is a common problem with fee-on-transfer tokens.

Recommendation

Consider implementing checks on minimum amounts to be transferred.

4.4 Fee Basis Points Can't Reach the Maximum Unlike What Is Described in the Documentation Minor ✓ Fixed

Resolution

Fixed in commit 0d22c5326e21541df0c718db98004d5a475aa2ea by setting `<=` instead of `<` on the `newBasisPoints` check in `setParams()`.

Description

The USDKG documentation describes that a fee on transfers may reach at most `0.2%`:

docs/USDKG.md?plain=1:L11

```
- Set fee parameters which, in basic terms, are equal to `0` and can reach a maximum of `0.2%`
```

However, the actual code implementation requires that the fee actually never reaches `0.2%` and is always strictly less than that:

contracts/USDKG.sol:L21

```
uint256 public constant MAX_BASIS_POINTS = 20;
```

contracts/USDKG.sol:L216-L223

```
function setParams(uint256 newBasisPoints) public onlyOwner {
    // ensure transparency by hardcoding limit beyond which fees can never be added
    require(newBasisPoints < MAX_BASIS_POINTS, "basis points should be less then MAX_BASIS_POINTS");

    basisPointsRate = newBasisPoints;

    emit Params(basisPointsRate);
}
```

Recommendation

Adjust code or documentation to be in sync.

4.5 Inaccurate Comments and Messages. Minor ✓ Fixed

Resolution
Fixed in commit 0d22c5326e21541df0c718db98004d5a475aa2ea by adjusting the relevant messages and comments.

Description

The codebase at times contains comments or messages that are inaccurate or have typos.

Examples

- `keccack(...` instead of `keccak256(...`:

contracts/Multisig.sol:L276-L277

```
function transferToken(address token, address receiver, uint256 amount) internal returns (bool transferred) {
    // 0xa9059cbb - keccack("transfer(address,uint256)")
```

- `Now owner` instead of `Not owner`:

contracts/Multisig.sol:L457

```
require(owners[msg.sender] != address(0), "Now owner");
```

- `@dev Throws if called by any account other than the owner.` instead of `@dev Throws if called by any account other than the **compliance**`.

contracts/USDKG.sol:L62-L65

```
/**
  * @dev Throws if called by any account other than the owner.
  */
modifier onlyCompliance() {
```

- The comment mentions `Tether` as is done in the original USDT contract, but the Tether entity is not applicable in the USDKG token:

contracts/USDKG.sol:L248-L249

```
// getters to allow the same blacklist to be used also by other contracts (including upgraded Tether)
function getBlackListStatus(address _maker) external view returns (bool) {
```

Recommendation

Adjust the codebase as appropriate.

4.6 Use `.call` to Transfer Native Token Refund Minor ✓ Fixed

Resolution
Fixed in commit 0d22c5326e21541df0c718db98004d5a475aa2ea by using `.call()` over `.send()`.

Description

Similarly to the original contracts that the `Multisig.sol` contract is built on - the Gnosis Safe multisig contract - it would be best to allow for a `.call()` to perform native token transfer for refunds in the `handlePayment()` function:

contracts/Multisig.sol:L257-L260

```
if (gasToken == address(0)) {
    // for ETH we will only adjust the gas price to not be higher than the actual used gas price
    payment = (gasUsed + baseGas) * (gasPrice < tx.gasprice ? gasPrice : tx.gasprice);
    require(receiver.send(payment), "Error when paying a transaction in native currency");
```

As is noted in Gnosis Safe multisig implementation's PR 601 and PR 602 from 2023, this does not introduce reentrancies but could allow for benefits, such as integrations with smart contract wallet refund recipients.

Recommendation

Adjust the refund for native tokens to use `.call()`.

4.7 Settle Solidity Version Minor ✓ Fixed

Resolution
Fixed in commit 0d22c5326e21541df0c718db98004d5a475aa2ea by setting the Solidity version to be `=0.8.24` in both contracts.

Description

The contracts `Multisig.sol` and `USDKG.sol` both have a free-floating `^0.8.0` Solidity version:

contracts/Multisig.sol:L2

```
pragma solidity ^0.8.0;
```

contracts/USDKG.sol:L2

```
pragma solidity ^0.8.0;
```

Recommendation

It would be best to settle on a specific version of Solidity before deployment to better predict performance and stability of the deployed contracts.

4.8 Inconsistent Transaction Execution Logic in Multisig Contract Deviates From Gnosis Safe and Documentation Minor ✓ Fixed

Resolution
Fixed in commit 0d22c5326e21541df0c718db98004d5a475aa2ea by allowing only owners to perform the execution of approved transactions.

Description

The `execTransaction` function in the Multisig contract allows any address to execute the function, provided the transaction is approved by the required threshold number of owners of the Multisig contract. The `checkApprovals` function checks if the transaction data hash (`txHash`) has enough approvals, if so the `execTransaction` proceeds to execute the transaction. This behavior deviates from the Gnosis Safe's implementation which requires that only the transaction approver can execute the transaction. This deviation may introduce some unknown risks to the system. Additionally the behavior also is inconsistent with the documentation of the Multisig contract which states

> The last signer initiates the transaction

Examples

contracts/Multisig.sol:L382-L437

```
function execTransaction(
    address to,
    uint256 value,
    bytes calldata data,
    Operation operation,
    uint256 safeTxGas,
    uint256 baseGas,
    uint256 gasPrice,
    address gasToken,
    address payable refundReceiver
) public payable virtual returns (bool success) {
    bytes32 txHash;
    // use scope here to limit variable lifetime and prevent `stack too deep` errors
    {
        bytes memory txHashData = encodeTransactionData(
            // transaction info
            to,
            value,
            data,
            operation,
            safeTxGas,
            // payment info
            baseGas,
            gasPrice,
            gasToken,
            refundReceiver,
            // signature info
            nonce
        );
        // increase nonce and execute transaction.
        nonce++;
        txHash = keccak256(txHashData);
        checkApprovals(txHash, txHashData);
    }
    // we require some gas to emit the events (at least 2500) after the execution and some to perform code until the execution (5
    // we also include the 1/64 in the check that is not send along with a call to counteract potential shortings because of EIP-
    require(gasleft() >= max((safeTxGas * 64) / 63, safeTxGas + 2500) + 500, "Insufficient gas");
    // use scope here to limit variable lifetime and prevent `stack too deep` errors
    {
        uint256 gasUsed = gasleft();
        // if the gasPrice is 0 we assume that nearly all available gas can be used (it is always more than safeTxGas)
        // we only substract 2500 (compared to the 3000 before) to ensure that the amount passed is still higher than safeTxGas
        success = execute(to, value, data, operation, gasPrice == 0 ? (gasleft() - 2500) : safeTxGas);
        gasUsed = gasUsed - gasleft();
        // if no safeTxGas and no gasPrice was set (e.g. both are 0), then the internal tx is required to be successful
        // this makes it possible to use `estimateGas` without issues, as it searches for the minimum gas where the tx doesn't re
        require(success || safeTxGas != 0 || gasPrice != 0, "Error during call");
        // we transfer the calculated tx costs to the tx.origin to avoid sending it to intermediate contracts that have made call
        uint256 payment = 0;
        if (gasPrice > 0) {
            payment = handlePayment(gasUsed, baseGas, gasPrice, gasToken, refundReceiver);
        }
        if (success) emit ExecutionSuccess(txHash, payment);
        else emit ExecutionFailure(txHash, payment);
    }
}
```

contracts/Multisig.sol:L146-L171

```
function checkApprovals(bytes32 dataHash, bytes memory data) public view {
    // load threshold to avoid multiple storage loads
    uint256 _threshold = threshold;
    // check that a threshold is set
    require(_threshold > 0, "Threshold is not set");
    checkNApprovals(dataHash, data, _threshold);
}

/**
 * @notice checks whether the signature provided is valid for the provided data and hash. Reverts otherwise
 * @dev since the EIP-1271 does an external call, be mindful of reentrancy attacks
 * @param dataHash hash of the data (could be either a message hash or transaction hash)
 * @param data that should be signed (this is passed to an external validator contract)
 * @param requiredSignatures amount of required valid signatures
 */
function checkNApprovals(bytes32 dataHash, bytes memory data, uint256 requiredSignatures) public view {
    uint256 count = 0;
    address currentOwner = owners[SENTINEL_OWNERS];
    while (currentOwner != SENTINEL_OWNERS) {
        if (approvedHashes[currentOwner][dataHash] != 0) {
            count++;
        }
        currentOwner = owners[currentOwner];
    }
    require(count >= requiredSignatures, "Not enough approvals");
}
```

Recommendation

Modify the transaction execution flow to ensure only transaction approver can execute the transaction

4.9 Unused Variable and Functions Minor ✓ Fixed

Resolution

Fixed in commit 0d22c5326e21541df0c718db98004d5a475aa2ea by removing unused functions and variables, namely `data` in `checkApprovals` and `checkNApprovals`, `signedMessages` mapping, and the internal `signatureSplit()` function.

Description

The `Multisig` contract contains several unused variables and functions, such as the `data` parameter in `checkApprovals` and `checkNApprovals` function, the `signedMessages` variable and the `signatureSplit` function. These unused elements add confusion and complexity to the code and contribute to increased gas cost.

Examples

contracts/Multisig.sol:L146-L171

```
function checkApprovals(bytes32 dataHash, bytes memory data) public view {
    // load threshold to avoid multiple storage loads
    uint256 _threshold = threshold;
    // check that a threshold is set
    require(_threshold > 0, "Threshold is not set");
    checkNApprovals(dataHash, data, _threshold);
}

/**
 * @notice checks whether the signature provided is valid for the provided data and hash. Reverts otherwise
 * @dev since the EIP-1271 does an external call, be mindful of reentrancy attacks
 * @param dataHash hash of the data (could be either a message hash or transaction hash)
 * @param data that should be signed (this is passed to an external validator contract)
 * @param requiredSignatures amount of required valid signatures
 */
function checkNApprovals(bytes32 dataHash, bytes memory data, uint256 requiredSignatures) public view {
    uint256 count = 0;
    address currentOwner = owners[SENTINEL_OWNERS];
    while (currentOwner != SENTINEL_OWNERS) {
        if (approvedHashes[currentOwner][dataHash] != 0) {
            count++;
        }
        currentOwner = owners[currentOwner];
    }
    require(count >= requiredSignatures, "Not enough approvals");
}
```

contracts/Multisig.sol:L47

```
mapping(bytes32 => uint256) public signedMessages;
```

contracts/Multisig.sol:L309-L323

```
function signatureSplit(bytes memory signatures, uint256 pos) internal pure returns (uint8 v, bytes32 r, bytes32 s) {
    // solhint-disable-next-line no-inline-assembly
    assembly {
        let signaturePos := mul(0x41, pos)
        r := mload(add(signatures, add(signaturePos, 0x20)))
        s := mload(add(signatures, add(signaturePos, 0x40)))
        /**
         * here we are loading the last 32 bytes, including 31 bytes
         * of 's'. There is no 'mload8' to do this
         * 'byte' is not working due to the Solidity parser, so lets
         * use the second best option, 'and'
         */
        v := and(mload(add(signatures, add(signaturePos, 0x41))), 0xff)
    }
}
```

Recommendation

Remove the unused variables and functions

Appendix 1 - Files in Scope

This audit covered the following files:

File	SHA-1 hash
contracts/Multisig.sol	`63a2b1443c5c36d56167af262925a2146f62b2f6`
contracts/USDKG.sol	`e14ee08519be2d4622f4c704bc379a0541371f57`

Appendix 2 - Disclosure

Consensys Diligence (“CD”) typically receives compensation from one or more clients (the “Clients”) for performing the analysis contained in these reports (the “Reports”). The Reports may be distributed through other means, including via Consensys publications and other distributions.

The Reports are not an endorsement or indictment of any particular project or team, and the Reports do not guarantee the security of any particular project. This Report does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. No Report provides any warranty or representation to any third party in any respect, including regarding the bug-free nature of code, the business model or

proprietors of any such business model, and the legal compliance of any such business. No third party should rely on the Reports in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset. Specifically, for the avoidance of doubt, this Report does not constitute investment advice, is not intended to be relied upon as investment advice, is not an endorsement of this project or team, and it is not a guarantee as to the absolute security of the project. CD owes no duty to any third party by virtue of publishing these Reports.

A.2.1 Purpose of Reports

The Reports and the analysis described therein are created solely for Clients and published with their consent. The scope of our review is limited to a review of code and only the code we note as being within the scope of our review within this report. Any Solidity code itself presents unique and unquantifiable risks as the Solidity language itself remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer, or any other areas beyond specified code that could present security risks. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. In some instances, we may perform penetration testing or infrastructure assessments depending on the scope of the particular engagement.

CD makes the Reports available to parties other than the Clients (i.e., “third parties”) on its website. CD hopes that by making these analyses publicly available, it can help the blockchain ecosystem develop technical best practices in this rapidly evolving area of innovation.

A.2.2 Links to Other Web Sites from This Web Site

You may, through hypertext or other computer links, gain access to web sites operated by persons other than Consensys and CD. Such hyperlinks are provided for your reference and convenience only, and are the exclusive responsibility of such web sites’ owners. You agree that Consensys and CD are not responsible for the content or operation of such Web sites, and that Consensys and CD shall have no liability to you or any other person or entity for the use of third party Web sites. Except as described below, a hyperlink from this web Site to another web site does not imply or mean that Consensys and CD endorses the content on that Web site or the operator or operations of that site. You are solely responsible for determining the extent to which you may use any content at any other web sites to which you link from the Reports. Consensys and CD assumes no responsibility for the use of third-party software on the Web Site and shall have no liability whatsoever to any person or entity for the accuracy or completeness of any outcome generated by such software.

A.2.3 Timeliness of Content

The content contained in the Reports is current as of the date appearing on the Report and is subject to change without notice unless indicated otherwise, by Consensys and CD.