Version 1.0 / 17.04.2018

kasko2go Token Contract Audit

inacta AG

Eugen Lechner <eugen.lechner@inacta.ch>
Cédric Walter <cedric.walter@inacta.ch>

Index

37. Introduction

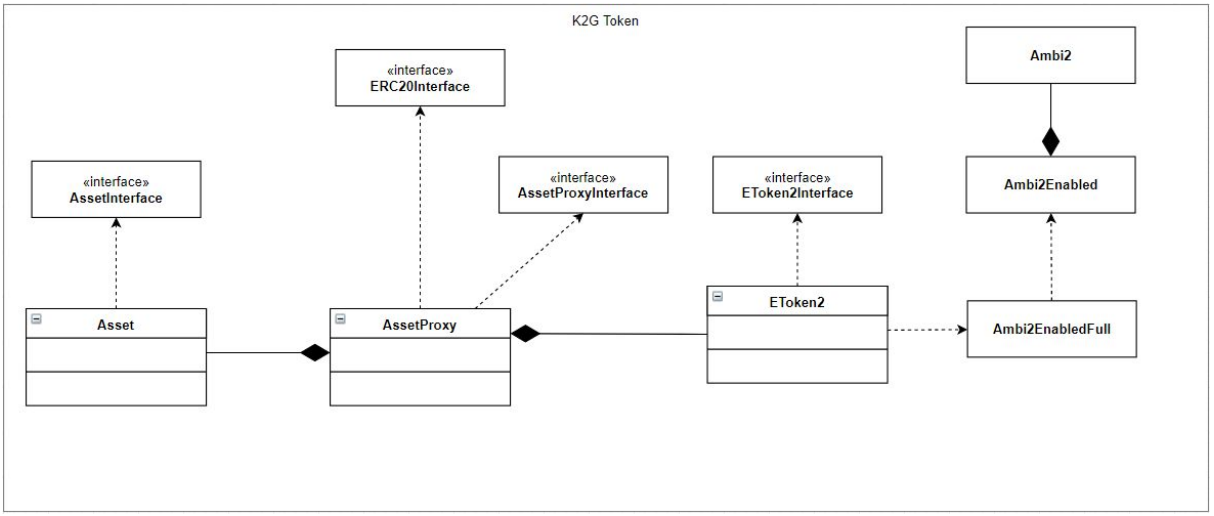
Name	kasko2go
Web Presence	https://kasko2go.com/
Name of Token	K2G
Whitepaper Reference	https://kasko2go.com/whitepaper.pdf
Smart Contract Code	K2GAudit.zip, received at 11.04.2018

38. Scope

This is an, 'eyes over' code review only of the contracts' source code. No static or dynamic testing have been done by the reviewing author. Only the Smart Contracts Code available in K2GAudit.zip have been reviewed.

The smart contract is only the ERC20 token. The entire token sales logic is located in the off-chain area of the ICO Platform and cannot be verified by this audit. The white paper specifications such as hardcap are therefore not part of this audit.

The audit was carried out between April 11rd and 12th, 2018 by using the following sources
- Paper "ICO White Paper"
- K2G Token Smart Contract Code (etoken2-contracts-907f9fa01b753845645c9985d2a04b45e5d23d43)



39. Executive Summary

We reviewed the K2G Token Sales Contract and did not find any critical security problems. Our findings included four medium and nine low issues. Most of the code does not comply with best practice recommendations, but generally leaves a solid

impression. It follows a generic approach with many proxy classes and integrated role management, which goes far beyond the ERC20 standard requirements.

Although token sales take place without a crowdsale smart contract, Ambisafe provides the necessary transparency for the distribution of tokens by executing all transactions on blockchain. It is the responsibility of kasko2go to distribute the tokens according to the information in white papers, there is no mechanism to automatically distribute the tokens. The conformity of crowdsale (number of tokens generated, allocations to founders) can been verified directly in ethereum blockchain.

40. Ambisafe

Ambisafe Inc. is an Ethereum Asset Platform based in San Francisco California.

Ambisafe is a platform that makes it easier for anyone to create blockchain-based digital assets. Using Ambisafe, anyone can issue a blockchain-based asset in minutes, then add it to cryptocurrency exchanges worldwide.

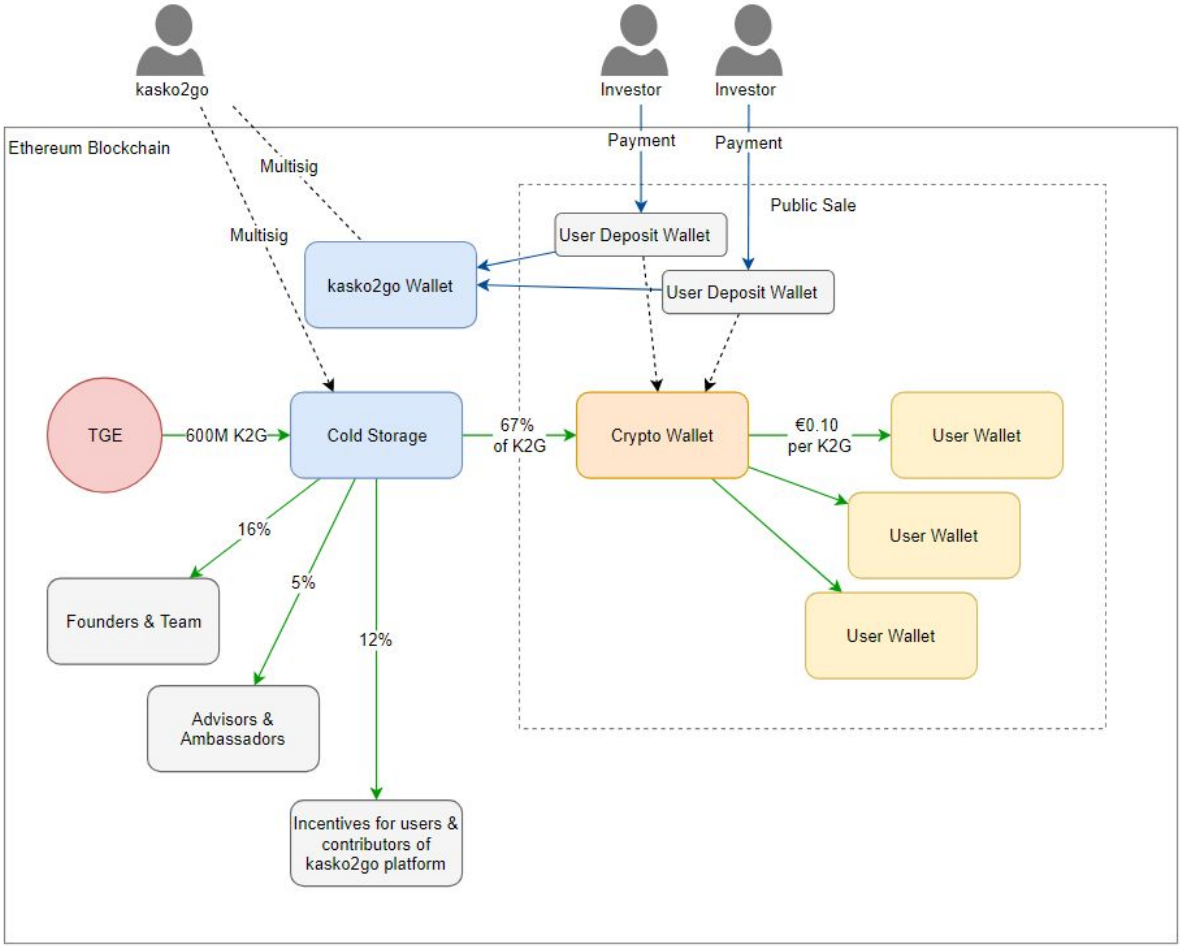
K2G tokens will be stored via the Ambisafe Asset Cold Storage with multi-sig capabilities which requires multiple confirmations from trusted parties to securely perform management actions.

Ambisafe offer an asset manager where it is possible to create the K2G assets to sell, a wallet to store and release them, a wallet to collect funds.

During ICO sale, investors purchase K2G tokens and their money is store in an online multisig Crypto-Wallet. Meanwhile, investors receive a generated own Wallets where they store the tokens they bought.

The following image shows how the individual token and payment transactions can be traced in the blockchain.

All tokens are generated at the start of ICO and transferred to the kasko2go Cold Storage.
From there it is the responsibility of kasko2go to distribute the tokens according to the information in white papers. This is performed manually by the ICO Managers of kasko2go distributing the tokens manually.
Some tokens are distributed directly (founders, team etc.). Tokens that will be distributed to the contributors in the K2G public token sale are moved to the automated Crypto Wallet and distributed from there according to the investor deposits at the daily rate.

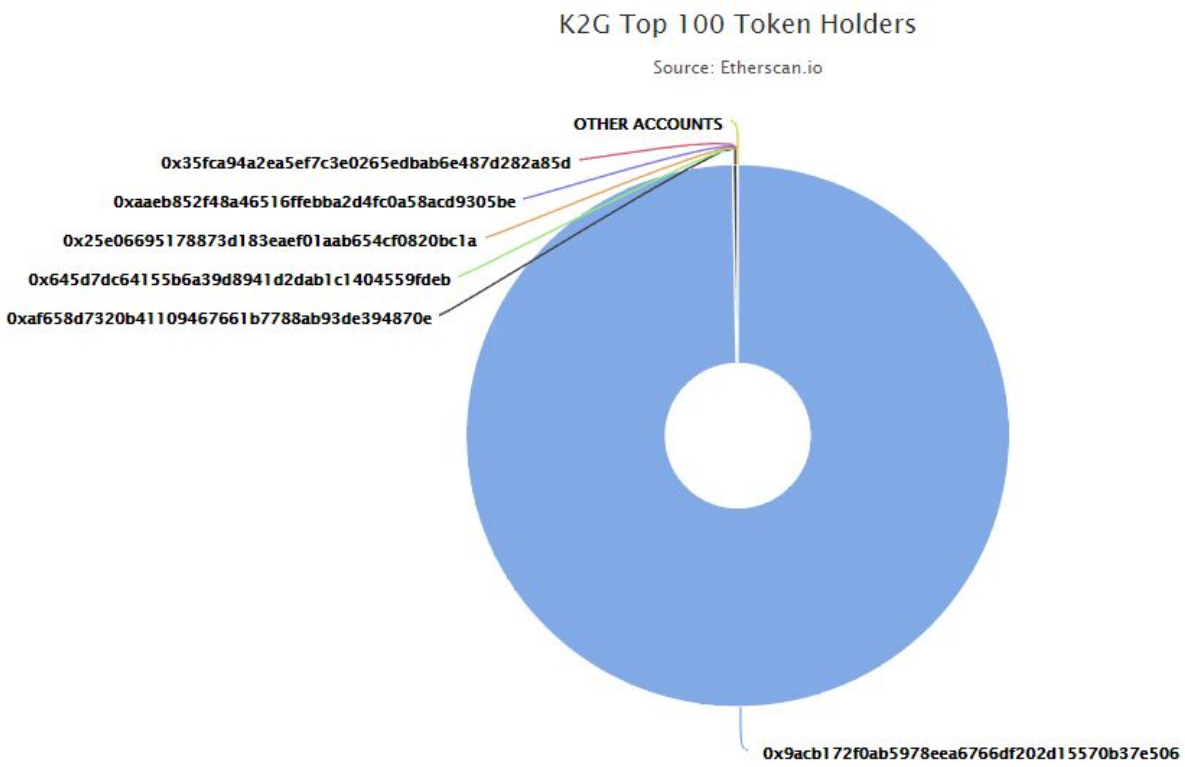


41. On chain analysis

K2G token can be seen at Ethereum address
https://etherscan.io/token/0x926703fb558f46331b6a06322bcf9e9d017fe6ec

From there we can certify that the number of Token is capped to 600 Millions tokens and match the whitepaper "kasko2go-whitepaper.pdf" (sha256sum c81946031618a167d2e2bb7e33213c4b2a238a915849569fc4f0d7b96f98ac48)

We can also see the distribution of wallets holding K2G tokens:



The wallet 0x9acb172f0ab5978eea6766df202d15570b37e506 is assumed to be the mutisig cold storage wallet holding 99.70% of K2G token.

The distribution of K2G tokens among founders, advisors, team as of today 20.04.2018 17:41 has not yet being done.

The deployed code at https://etherscan.io/address/0x926703fb558f46331b6a06322bcf9e9d017fe6ec#code

42. Findings

Here is our assessment and recommendations.

	Description	Severity	Priority
1	Using one year old Solidity compiler version (6.1)	Medium	Medium
2	High code complexity (6.4.1)	Low	Low
3	Use safe math operators (6.4.2)	Medium	Medium
4	Avoid negated conditions (6.4.4)	Low	Low
5	Patterns for error handling (6.4.5)	Medium	Medium
6	Event name consistency, invoking events without "emit" prefix is deprecated. (6.7)	Low	Low
7	Explicitly mark visibility in function (6.4.10)	Low	Low

8	Avoid to make time-based decisions in your business logic. (6.4.11)	Medium	Medium
9	Avoid commented code (6.4.12)	Low	Low
10	Missing circuit breaker (6.4.13)	Low	Low
11	Prefer newer solidity constructs (6.4.14)	Low	Low
12	Many Typography errors (6.4.15)	Low	Low
13	Lock pragmas to specific compiler version (6.4.16)	Low	Low

43. Review Report of Token Sale Smart Contract

43.1 Use latest Solidity version

The Smart Contract Code is compiled with different versions of Solidity.

- The EToken2 contract uses the Solidity compiler version 0.4.8 from Jan 13, 2017. Since this version many security fixes and improvements in the compiler have been identified and implemented.
- The Asset Contract uses the Solidity compiler version 0.4.15 from Aug 8, 2017.

We strongly recommend using the latest stable version 0.4.21 for all smart contracts to take advantage of the latest features of Solidity and to avoid potential security risks.

The version of Truffle has to match the compiler version of solidity. Since you are using Truffle 3.1.9, we recommend you to update to Truffle 4.1.5

from

```
"dependencies": {
 "ethereumjs-testrpc": "3.0.3",
 "truffle": "3.1.9"
},
```

to

```
"dependencies": {
 "ethereumjs-testrpc": "3.0.3",
 "truffle": "4.1.5"
},
```

43.2 Compliance with ERC20 Standard

The K2G Token implements the ERC20 Token Standard Interface. The generating of asset through the platform Ambisafe should be like all other assets generated before. We have no access to the off-chain platform and its source code and therefore cannot make any statement as to whether the sales process will be complied with white paper.

43.3 Compliance with Token Sale Terms and Conditions

The conditions controlling the Token Sale is located in the off-chain at the ICO Platform and cannot be verified by this audit. All payments made by investors cannot be verified in Blockchain and must be verified in off-chain platform. We recommend

carrying out an independent audit of the generated tokens after the end of the sales process.

43.4 Compliance with Smart Contract Best Practices

43.4.1 General code checking

Findings

The code implements many interfaces and contains a lot of logic that goes far beyond the ERC20 standard. Many sections are not in accordance with current best practice recommendations, such as no error are throw in case of wrong parameters in order to reduce the caller's gas costs. We recommend keep the contracts small and easily understandable.

Recommendations

In Solidity 0.4.10 require() were introduced. require(condition) is meant to be used for input validation, which should be done on any user input, and reverts if the condition is false.

As such we recommend you to replace all code pattern like these

```
modifier checkEnabledSwitch(bytes32 _switch) {
   if (!isEnabled(_switch)) {
       _error('Feature is disabled');
   } else {
       _;
   }
}
```

with

```
modifier checkEnabledSwitch(bytes32 _switch) {
   required(!isEnabled(_switch))
   _;
}
```

43.4.2 Use safe math operators

Findings

In order to remove the risk of overflow and underflow when using most mathematical operations, we recommend using the SafeMath library like the one provided by OpenZeppelin.

Over and under flows An overflow happens when the limit of the type variable uint256 , 2 ** 256, is exceeded. What happens is that the value resets to zero instead of incrementing more.

For instance, if I want to assign a value to a uint bigger than 2 ** 256 it will simple go to 0 — this is dangerous.

On the other hand, an underflow happens when you try to subtract from 0 a number bigger than 0.For example, if you subtract 0 - 1 the result will be = 2 ** 256 instead of -1.

This is quite dangerous. For example, in the Smart Contract MultiAsset.sol:

```
function _transferDirect(uint _fromId, uint _toId, uint _value, bytes32 _symbol)
internal {
   assets[_symbol].wallets[_fromId].balance -= _value;
   assets[_symbol].wallets[_toId].balance += _value;
}
```

If the parameter _value is a negative value, the balance at the caller is increased instead of decreased.

uint256 is only used in interface ERC20Interface.sol but we don't see the implementation of it, Ambisafe has the responsibility to ensure that they are using a SafeMath library.

Recommendations

We recommend you to reuse (not copy...) the implementation of Open Zeppelin (link) which is well reviewed and battle tested. The latest version as of today is "zeppelin-solidity": "1.8.0", here it is as reference

```
library SafeMath {

 /**
 * @dev Multiplies two numbers, throws on overflow.
 */
 function mul(uint256 a, uint256 b) internal pure returns (uint256) {
   if (a == 0) {
     return 0;
   }
   uint256 c = a * b;
   assert(c / a == b);
   return c;
 }

 /**
 * @dev Integer division of two numbers, truncating the quotient.
```

```
  */
  function div(uint256 a, uint256 b) internal pure returns (uint256) {
    // assert(b > 0); // Solidity automatically throws when dividing by 0
    uint256 c = a / b;
    // assert(a == b * c + a % b); // There is no case in which this doesn't hold
    return c;
  }

  /**
  * @dev Subtracts two numbers, throws on overflow (i.e. if subtrahend is greater
than minuend).
  */
  function sub(uint256 a, uint256 b) internal pure returns (uint256) {
    assert(b <= a);
    return a - b;
  }

  /**
  * @dev Adds two numbers, throws on overflow.
  */
  function add(uint256 a, uint256 b) internal pure returns (uint256) {
    uint256 c = a + b;
    assert(c >= a);
    return c;
  }
}
```

43.4.3 Use modifiers for recurring checks

Findings

No Findings. All contracts use modifiers very proficiently. The use of modifiers in the functions and state variables are explicitly specified. This increases the readability of the contract and makes it more trustworthy.

43.4.4 Avoid negated conditions

Findings

The negated conditions can cause errors if the condition is complex and must be avoided.

Recommendations

The smart contracts contains many negated conditions, some of them can be simplified. E.g.: `!isEnabled` can be used as `isDisabled`

43.4.5 Patterns for error handling

Findings

The Smart Contract token does not throw errors when input parameters is not valid.

Recommendations

This means that transactions with incorrect parameters are still accepted from Blockchain with the following disadvantages:

- The gas fees of callers are consumed even if it has not led to a result
- The state of the Smart Contract can still change, although it should not lead to any changes if the input parameters are incorrect. The Ethereum EVM guarantees that the state is unchanged only if a error is thrown.

43.4.6 Use view/constant modifier if possible

Findings

No findings. The smart contracts use constant modifier if possible.

43.4.7 Implements default payable function

Findings

No findings.

43.4.8 Token Lifecycle

Findings

No findings.

43.4.9 Crowdsale Lifecycle

Findings

The entire sales life cycle is controlled outside the blockchain and cannot be verified by this audit.

43.4.10 Explicitly mark visibility in function

Findings

Solidity defaults to public. Consider making this explicit.

Recommendations

This will also avoid some compiler warnings introduced in the latest versions.

for example in EToken2.sol

```
mapping(bytes32 => bool) switches;
mapping(address => uint) holderIndex;
```

43.4.11 Avoid to make time-based decisions in your business logic

Findings

The timestamp of the block can be manipulated by the miner, and all direct and indirect uses of the timestamp should be considered.Block numbers and average block time can be used to estimate time.
for example in AssetProxy.sol L. 464/497

Recommendations

Date-related logic is not critical, and as such code can be kept like that.

43.4.12 Avoid commented code

Findings

Because of commented code certain portion of the contract is difficult to understand and give the feeling that the contract is not ready yet for deployment.

RegistryICAP.sol L. 61/62 & 85/88

Recommendations

Use Git or any code versioning software to avoid having commended code spread in shippable code.

43.4.13 Missing circuit breaker

Findings

It will be good if contract will have a circuit breaker to stop contract execution in case of any emergency or attacks.

Recommendations

see
https://consensys.github.io/smart-contract-best-practices/software_engineering/#circuit-breakers-pause-contract-functionality.

43.4.14 Prefer newer solidity constructs

Findings

Prefer constructs/aliases such as keccak256 (over sha3). While not a security issue, sha3 is deprecated in Ethereum and do not behave like sha3 introducing confusion when external system would calculate sha3 differently.

Recommendations

Replace sha3 with keccak-256 to express the reality to potential users of your APIs.

Ethereum uses KECCAK-256. It should be noted that it does not follow the FIPS-202 based standard of Keccak, which was finalized in August 2015.

Hashing the string "testing":

- *Ethereum SHA3 function in Solidity = 5f16f4c7f149ac4f9510d9cf8cf384038ad348b3bcdc01915f95de12df9d1b02*
- *Keccak-256 (Original Padding) = 5f16f4c7f149ac4f9510d9cf8cf384038ad348b3bcdc01915f95de12df9d1b02*
- *SHA3-256 (NIST Standard) = 7f5979fb78f082e8b1c676635db8795c4ac6faba03525fb708cb5fd68fd40c5e*

see
https://consensys.github.io/smart-contract-best-practices/recommendations/#prefer-newer-solidity-constructs
According to ethereum/EIPs#59, solidity has already been updated

https://github.com/ethereum/solidity/blob/develop/Changelog.md#0417-2017-09-21
https://github.com/ethereum/solidity/pull/1164/files

http://ethereum.stackexchange.com/questions/550/which-cryptographic-hash-function-does-ethereum-use

And the Yellowpaper also mentions it at every occasion except for the OP-code name itself.

43.4.15 Many typography error

Findings

Typos make the code difficult to read and understand.

In EToken2.sol

```
Sets EventsHstory contract address. -> Sets EventsHistory contract address.
existance -> existence
```

In AssetProxy.sol

```
Timespan -> TimeSpan
etoken2 -> eToken2
Prforms -> Performs
event UpgradeCommited(address newVersion);  -> event UpgradeCommitted(address
newVersion);
```

Recommendations
Correct typos.

43.4.16 Lock pragmas to specific compiler version

Findings
Contracts should be deployed with the same compiler version and flags that they have been tested the most with. Locking the pragma helps ensure that contracts do not accidentally get deployed using, for example, the latest compiler which may have higher risks of undiscovered bugs. Contracts may also be deployed by others and the pragma indicates the compiler version intended by the original authors.

Recommendations
In Ambi.sol L. 1

```
// bad
pragma solidity ^0.4.8;


// good
pragma solidity 0.4.8;
```

43.5 Compliance with Code Style

Findings
No findings. Smart contracts code corresponds for the most part to the recommended Code Style. It does not contain any complex duplicate code that can lead to diverging program logic.

43.6 Missing Input Validation

Findings
No findings. *Unexpected method arguments may result in insecure contract behaviors. To avoid this, contracts must check whether all transaction arguments meet their desired preconditions.*

43.7 Avoid compiler warnings

Findings
If you start using the latest compiler…

Recommendations

we recommend to use ”emit” prefix in all event names to avoid following compiler warnings:

Warning: Invoking events without "emit" prefix is deprecated.

43.8 Check known security attacks

Findings

No findings. The patterns of attacks that have already been successfully executed are checked.

43.8.1 ERC20 API: An Attack Vector on Approve/TransferFrom Methods

Findings

No findings. *To prevent attack vectors clients should make sure to create user interfaces in such a way that they set the allowance first to 0 before setting it to another value for the same spender. Though the contract itself shouldn't enforce it, to allow backwards compatibility with contracts deployed before.*

43.8.2 Re-Entrancy (Checks-Effects-Interactions Pattern)

Findings

No findings in smart contract code.

43.8.3 Transactions May Affect Ether Receiver

Findings

No findings. *A contract is exposed to this vulnerability if a miner (who executes and validates transactions) can reorder the transactions within a block in a way that affects the receiver of ether.*

43.8.4 Unhandled Exception

Findings

No findings. *A call/send instruction returns a non-zero value if an exception occurs during the execution of the instruction (e.g., out-of-gas). A contract must check the return value of these instructions and throw an exception.*

Limitations

Only the Solidity source code of the smart contract has been reviewed and audited. The low-level assembly code that is created by the Solidity compiler was excluded from the review. Solely the token smart contract has been reviewed; the ICO platform in any other matters have not been part of the assessment.