Kambria DAOs Dev Payment Module

Smart Contract Security Audit

Prepared by ShellBoxes

September 18th, 2023 - September 21st, 2023

Shellboxes.com

contact@shellboxes.com

Document Properties

Client	Kambria
Version	1.0
Classification	Public

Scope

Contract Name	Contract Address
DevPayment	0xDC402a1cBB34D52D18636B8ffab82dEc8D3296c0
DevPayment	0xc09746D96f44Fc6115ed0891BC32eD83B90F99DE

Re-Audit

Contract Name	Contract Address
DevPayment	0x9Cf909ec3AB2ACe345C2c4f58824dec92974C540

Contacts

COMPANY	EMAIL
ShellBoxes	contact@shellboxes.com

Contents

1 Introduction

Kambria engaged ShellBoxes to conduct a security assessment on the Kambria DAOs Dev Payment Module beginning on September 18th, 2023 and ending September 21st, 2023. In this report, we detail our methodical approach to evaluate potential security issues associated with the implementation of smart contracts, by exposing possible semantic discrepancies between the smart contract code and design document, and by recommending additional ideas to optimize the existing code. Our findings indicate that the current version of smart contracts can still be enhanced further due to the presence of many security and performance concerns.

This document summarizes the findings of our audit.

1.1 About Kambria

Kambria is an open innovation platform for Deep Tech (AI, Robotics, Blockchain, VR/AR, IoT...).Via their platform especially with Kambria DAOs, anyone can collaborate in researching, developing and commercializing deeptech solutions and get rewarded fairly for their contributions.

Issuer	Kambria
Website	`https://kambria.io`
Type	Solidity Smart Contract
Documentation	Kambria DAOs Dev Payment Module Brief
Audit Method	Whitebox

1.2 Approach & Methodology

ShellBoxes used a combination of manual and automated security testing to achieve a balance between efficiency, timeliness, practicability, and correctness within the audit's scope. While manual testing is advised for identifying problems in logic, procedure, and implementation, automated testing techniques help to expand the coverage of smart contracts and can quickly detect code that does not comply with security best practices.

1.2.1 Risk Methodology

Vulnerabilities or bugs identified by ShellBoxes are ranked using a risk assessment technique that considers both the LIKELIHOOD and IMPACT of a security incident. This framework is effective at conveying the features and consequences of technological vulnerabilities.

Its quantitative paradigm enables repeatable and precise measurement, while also revealing the underlying susceptibility characteristics that were used to calculate the Risk scores. A risk level will be assigned to each vulnerability on a scale of 5 to 1, with 5 indicating the greatest possibility or impact.

- Likelihood quantifies the probability of a certain vulnerability being discovered and exploited in the untamed.
- Impact quantifies the technical and economic costs of a successful attack.
- Severity indicates the risk's overall criticality.

Probability and impact are classified into three categories: H, M, and L, which correspond to high, medium, and low, respectively. Severity is determined by probability and impact and is categorized into four levels, namely Critical, High, Medium, and Low.

Impact	High	Critical	High	Medium
	Medium	High	Medium	Low
	Low	Medium	Low	Low
		High	Medium	Low

Likelihood

2 Findings Overview

2.1 Summary

The following is a synopsis of our conclusions from our analysis of the Kambria DAOs Dev Payment Module implementation. During the first part of our audit, we examine the smart contract source code and run the codebase via a static code analyzer. The objective here is to find known coding problems statically and then manually check (reject or confirm) issues highlighted by the tool. Additionally, we check business logics, system processes, and DeFi-related components manually to identify potential hazards and/or defects.

2.2 Key Findings

In general, these smart contracts' implementation might be improved by addressing the discovered flaws, which include 2 critical-severity, 1 high-severity, 1 medium-severity, 2 low-severity vulnerabilities.

Vulnerabilities	Severity	Status
SHB.1. Significant Conversion Error Due to Ignored Token Decimals	CRITICAL	Fixed
SHB.2. Potential Zero Payment Impact Due to Untrusted Exchange Rate Input	CRITICAL	Mitigated
SHB.3. Mismatch Between Stated Percentages and Actual Allocations	HIGH	Fixed
SHB.4. Excessive Reliance on Admin for Exchange Rate Input	MEDIUM	Mitgated
SHB.5. Missing Input Checks in Constructor	LOW	Fixed
SHB.6. Use of Floating Pragma Statement	LOW	Fixed

3 Finding Details

SHB.1 Significant Conversion Error Due to Ignored Token Decimals

- Severity : CRITICAL
- Status : Fixed
- Likelihood : 3
- Impact : 3

Description:

The contract fails to account for the differing decimals between KAT (18 decimals) and USDT (6 decimals) when performing conversions between the two. This oversight can lead to significant errors in the conversion rate, especially when relying on accurate inputs from the Client.

Exploit Scenario:

Consider an example where 1 USDT is equivalent to 100 KAT. Taking into account the decimals, this translates to 1e6 units of USDT being equivalent to 1e20 units of KAT. If a well-intentioned Client uses these values in the completeMilestone function, the resulting exchange rate becomes $\frac{10^6}{10^{20}}$ or $\frac{1}{10^{14}}$. This is a drastic deviation from the intended rate of $\frac{1}{100}$ or $\frac{1}{10^2}$. Such a discrepancy can lead to gross misallocations of funds.

Files Affected:

SHB.1.1: DevPayment.sol

```
uint256 katBalanceOnUSDT = (IERC20(katAddress).balanceOf(
    address(daoAddress)
) * _USDTAmount) / _EquivalentKATPerUSDT;
```

SHB.1.2: DevPayment.sol

```
uint256 katAmount = (milestoneAmount *
```

```
katBalanceOnUSDT *
_EquivalentKATPerUSDT) /
_USDTAmount /
totalBalanceOnUSDT;
```

Recommendation:

To address this issue, the contract should normalize the decimals of the tokens before performing any conversions. This can be achieved by multiplying or dividing the amounts by the difference in decimals to ensure that the conversion rate is consistent with the actual market rate. Implementing this normalization will ensure accurate and fair conversions between KAT and USDT.

Updates

The team resolved the issue by normalizing the decimals when converting the token amounts.

SHB.1.3: DevPayment.sol

```
uint256 katBalanceOnUSDTWei =
    ((BEP20Token(katAddress).balanceOf(address(daoAddress)) *
        _USDTAmount) *(10**usdtDecimals)/
    ((10**katDecimals)*_EquivalentKATPerUSDT));
```

SHB.1.4: DevPayment.sol

```
uint256 katAmount = (milestoneAmount *
    katBalanceOnUSDTWei *
    _EquivalentKATPerUSDT * (10 ** katDecimals)) /
    (_USDTAmount *totalBalanceOnUSDTWei* (10**usdtDecimals));
```

SHB.2 Potential Zero Payment Impact Due to Untrusted Exchange Rate Input

- Severity: CRITICAL
- Status: Mitigated
- Likelihood: 3
- Impact: 3

Description:

In the completeMilestone function, the contract relies on _USDTAmount and _EquivalentKATPerUSDT provided by the Client to determine the exchange rate between KAT and USDT. Since the Client role is not inherently trusted, there's no guarantee that the provided exchange rate is accurate. This design flaw can lead to potential manipulation of the actual payment amounts.

Exploit Scenario:

A malicious Client can exploit this by setting _EquivalentKATPerUSDT to a very low value (e.g., 1) and _USDTAmount to an exceedingly high value. This will artificially inflate the totalBalanceOnUSDT, causing both katAmount and usdtAmount to round down to zero due to integer division. As a result, when a milestone is marked as completed, the contractor will not receive any KAT or USDT funds. If this is repeated for all milestones, the contractor ends up receiving no compensation for their work, even though the milestones are marked as completed.

Files Affected:

SHB.2.1: DevPayment.sol

```
function completeMilestone(
    uint256 _milestoneIndex,
    uint256 _USDTAmount,
    uint256 _EquivalentKATPerUSDT
) public onlyClient {
```

```
require(_milestoneIndex < totalMilestones, "Invalid milestone index
    ↪ ");
require(
    !milestones[_milestoneIndex].completed,
    "Milestone has already been completed"
);

milestones[_milestoneIndex].completed = true;
completedMilestones++;
emit MilestoneCompleted(_milestoneIndex);

uint256 katBalanceOnUSDT = (IERC20(katAddress).balanceOf(
    address(daoAddress)
) * _USDTAmount) / _EquivalentKATPerUSDT;
uint256 usdtBalance = IERC20(usdtAddress).balanceOf(
    address(daoAddress)
);
uint256 totalBalanceOnUSDT = usdtBalance + katBalanceOnUSDT;

uint256 milestoneAmount = milestones[_milestoneIndex].amount;

// Calculate 30% of the milestone amount
uint256 katAmount = (milestoneAmount *
    katBalanceOnUSDT *
    _EquivalentKATPerUSDT) /
    _USDTAmount /
    totalBalanceOnUSDT;

// Calculate 70% of the milestone amount
uint256 usdtAmount = (milestoneAmount * usdtBalance) /
    totalBalanceOnUSDT;
```

Recommendation:

To mitigate this vulnerability, the contract should fetch the exchange rate from a trusted oracle or a reliable decentralized price feed. This ensures that the exchange rate used is accurate and not subject to manipulation by any party. Additionally, consider adding checks to prevent the possibility of katAmount and usdtAmount rounding down to zero.

Updates

The team addressed the problem by limiting the completeMilestone function to the admin role. The project views the admin as a reliable role, expecting them to input accurate and current exchange rates. Nonetheless, it's advisable to introduce a test case for the completeMilestone function. This ensures that the _EquivalentKATPerUSDT value inputted by the admin aligns with the required format. Even a valid exchange rate can lead to significant issues if not formatted as the function anticipates.

SHB.2.2: DevPayment.sol

```
function completeMilestone(
    uint256 _milestoneIndex,
    uint256 _USDTAmount,
    uint256 _EquivalentKATPerUSDT
) public onlyAdmin() {
```

SHB.3 Mismatch Between Stated Percentages and Actual Allocations

- Severity : HIGH
- Status : Fixed
- Likelihood : 3
- Impact : 2

Description:

The comments in the completeMilestone function suggest that the katAmount is calculated as 30% of the milestone amount and the usdtAmount is calculated as 70% of the milestone amount. However, the actual calculations depend on the USDT and KAT balances of the DAO contract, which may not necessarily align with the stated 30% and 70% allocations.

Files Affected:

SHB.3.1: DevPayment.sol

```
// Calculate 30% of the milestone amount
uint256 katAmount = (milestoneAmount *
    katBalanceOnUSDT *
    _EquivalentKATPerUSDT) /
    _USDTAmount /
    totalBalanceOnUSDT;

// Calculate 70% of the milestone amount
uint256 usdtAmount = (milestoneAmount * usdtBalance) /
    totalBalanceOnUSDT;
```

Recommendation:

Ensure that the DAO contract's balances of KAT and USDT are maintained in a way that reflects the intended distribution. Alternatively, consider revising the calculation method to guarantee the 30% and 70% allocations, regardless of the DAO contract's balances. Additionally, update the comments to accurately reflect the implemented logic.

Updates

The team resolved the issue by removing the comments, stating that the allocation will not be fixed to 30% and 70%, and it will depend on the USDT and KAT balances of the DAO.

SHB.4 Excessive Reliance on Admin for Exchange Rate Input

- Severity : MEDIUM
- Status : Mitgated
- Likelihood : 1
- Impact : 3

Description:

In the second version of the contract, the completeMilestone function in the contract places an undue amount of trust in the admin to provide the exchange rate between USDT and KAT. This design choice introduces several risks:

- If the admin key is compromised, malicious actors can manipulate the exchange rate to their advantage.
- Even well-intentioned admins can make mistakes, inputting incorrect exchange rates that could lead to financial discrepancies.
- The function lacks mechanisms to verify the accuracy of the exchange rate, potentially leading to unfair conversions and financial losses for stakeholders.

Files Affected:

SHB.4.1: DevPayment.sol

```
function completeMilestone(
    uint256 _milestoneIndex,
    uint256 _USDTAmount,
    uint256 _EquivalentKATPerUSDT
) public onlyAdmin() {
    require(_milestoneIndex < totalMilestones, "Invalid milestone index
        ↪ ");
    require(
        !milestones[_milestoneIndex].completed,
        "Milestone has already been completed"
    );
```

```
milestones[_milestoneIndex].completed = true;
completedMilestones++;
emit MilestoneCompleted(_milestoneIndex);

uint8 katDecimals = BEP20Token(katAddress).decimals();
uint8 usdtDecimals = BEP20Token(usdtAddress).decimals();

uint256 katBalanceOnUSDTWei =
    ((BEP20Token(katAddress).balanceOf(address(daoAddress)) *
        _USDTAmount) /
    _EquivalentKATPerUSDT)
    *(10**usdtDecimals)/(10**katDecimals);

uint256 usdtWeiBalance = BEP20Token(usdtAddress).balanceOf(address(
    ↪ daoAddress));
uint256 totalBalanceOnUSDTWei = usdtWeiBalance + katBalanceOnUSDTWei
    ↪ ;

uint256 milestoneAmount = milestones[_milestoneIndex].amount;

uint256 katAmount = ((milestoneAmount *
    katBalanceOnUSDTWei *
    _EquivalentKATPerUSDT) /
    _USDTAmount /
    totalBalanceOnUSDTWei)
     * (10 ** katDecimals) / (10**usdtDecimals);

uint256 usdtAmount = (milestoneAmount * usdtWeiBalance) /
    totalBalanceOnUSDTWei;
```

Recommendation:

To mitigate the risks associated with the admin's excessive power and potential for input errors, consider the following improvements:

- Consider introducing multi-signature requirements or a voting mechanism for critical actions like setting exchange rates. This distributes power and decision-making, reducing single points of failure and the potential for errors or malicious actions.
- Implement a mechanism to fetch the exchange rate from a trusted external source or oracle. This reduces the reliance on manual admin input and provides a more accurate and up-to-date rate.

Updates

The team mitigated the risk by implementing a Safe multi-signature wallet with 2 out of 2 threshold as the admin to reduce the centralization risk.

SHB.5 Missing Input Checks in Constructor

- Severity : LOW
- Status : Fixed
- Likelihood : 1
- Impact : 2

Description:

The constructor of the DevPayment contract initializes several critical variables, including addresses for the client, project manager, contractor, DAO, USDT, and KAT. However, there are no input checks to ensure that these addresses are valid or distinct from each other.

Files Affected:

SHB.5.1: DevPayment.sol

```
constructor(
    address _client,
```

```
    address _projectManager,
    address _contractor,
    IDao _daoAddress,
    IERC20 _usdtAddress,
    IERC20 _katAddress
) {
    client = _client;
    projectManager = _projectManager;
    contractor = _contractor;
    daoAddress = _daoAddress;
    usdtAddress = _usdtAddress;
    katAddress = _katAddress;
}
```

Recommendation:

Implement input validation checks in the constructor to ensure:

1. None of the provided addresses are zero addresses.
2. Critical roles like client, projectManager, and contractor are distinct.
3. The provided addresses for DAO, USDT, and KAT are valid contract addresses.

Updates

The team resolved the issue by implementing the required input checks.

SHB.5.2: DevPayment.sol

```
constructor(
    address _admin,
    address _projectManager,
    address _contractor,
    IDao _daoAddress,
    BEP20Token _usdtAddress,
    BEP20Token _katAddress
```

```
) {
    // Check that none of the provided addresses are zero addresses.
    require(_admin != address(0x00), "Admin address cannot be zero");
    require(
        _projectManager != address(0x00),
        "Project Manager address cannot be zero"
    );
    require(
        _contractor != address(0x00),
        "Contractor address cannot be zero"
    );
    require(
        address(_daoAddress) != address(0x00),
        "DAO address cannot be zero"
    );
    require(
        address(_usdtAddress) != address(0x00),
        "USDT address cannot be zero"
    );
    require(
        address(_katAddress) != address(0x00),
        "KAT address cannot be zero"
    );

    // Check that the provided addresses for DAO, USDT, and KAT are
        ↪ valid contract addresses.

    require(
        checkValidAddress(address(_daoAddress)),
        "DAO address is not a valid contract address"
    );
    require(
        checkValidAddress(address(_usdtAddress)),
        "USDT address is not a valid contract address"
```

```
    );
    require(
        checkValidAddress(address(_katAddress)),
        "KAT address is not a valid contract address"
    );

    admin = _admin;
    projectManager = _projectManager;
    contractor = _contractor;
    daoAddress = _daoAddress;
    usdtAddress = _usdtAddress;
    katAddress = _katAddress;
}
```

SHB.6 Use of Floating Pragma Statement

- Severity : LOW
- Status : Fixed
- Likelihood : 1
- Impact : 1

Description:

The contract uses a floating pragma statement, which indicates that it can be compiled with any Solidity compiler version from 0.8.0 (inclusive) up to, but not including, version 0.9.0. While this provides flexibility, it can also introduce risks if the contract is compiled with a newer compiler version that contains breaking changes or unexpected behaviors.

Files Affected:

SHB.6.1: DevPayment.sol

```
pragma solidity ^0.8.0;
```

Recommendation:

Specify a fixed compiler version in the pragma statement to ensure consistent behavior and avoid potential pitfalls introduced by newer compiler versions. For instance, if the contract was tested and audited using Solidity version 0.8.4 for example, then use pragma solidity 0.8.4; to lock in that specific version.

Updates

The team resolved the issue by fixing the pragma version to 0.8.13.

4 Best Practices

BP.1 Remove Unnecessary Initializations

Description:

The contract explicitly initializes the variable totalMilestoneAmount with a default value of 0. In Solidity, state variables are automatically initialized to their default values. For uint256, this default is 0. This explicit setting is redundant and can make the code longer without adding any functional benefit. It's recommended to rely on Solidity's default initialization to make the code cleaner and more concise.

Files Affected:

BP.1.1: DevPayment.sol

```
uint256 totalMilestoneAmount = 0;
```

Status - Fixed

BP.2 Optimize Loops for Efficiency

Description:

The loop iterating over totalMilestones can be optimized for better efficiency. Caching totalMilestones into memory reduces the gas cost associated with repeatedly accessing a state variable. Additionally, using pre-increments inside an unchecked block can further reduce gas costs by avoiding overflow checks. Lastly, the loop variable i can be declared without an explicit initialization to 0, as it's the default value for uint256. It's recommended to implement these optimizations to enhance the contract's efficiency and reduce gas consumption.

Files Affected:

BP.2.1: DevPayment.sol

```
for (uint256 i = 0; i < totalMilestones; i++) {
    totalMilestoneAmount += milestones[i].amount;
}
```

Status - Fixed

5 Tests

Results:

→ DevPayment

✓ Should get the right Admin Role

✓ Should get the right Project Manager Role

✓ Should get the right Contractor Role

✓ Should get the right DAO Address

✓ Should get the right USDT Address

✓ Should get the right KAT Address

✓ Should get the right total payment

✓ Should get the right total mile stone

✓ Should get the right mile stone amounts

(9 passed)

Conclusion:

The project offers a testing mechanism to improve the correctness of smart contracts; nonetheless, we advise increasing the numbers of scenarios and tests to cover all functionalities and edge cases in order to guarantee the integrity of the code and the functionality of the contract.

6 Conclusion

In this audit, we examined the design and implementation of Kambria DAOs Dev Payment Module contract and discovered several issues of varying severity. Kambria team addressed 4 and mitigated 2 issues raised in the initial report implementing the necessary fixes. Shellboxes' auditors advised Kambria Team to maintain a high level of vigilance and to keep the mitigated findings in mind in order to avoid any future complications.

7 Scope Files

7.1 Audit

Files	MD5 Hash
DevPayment.sol	fedd40397ff15b88425ca59b65ff1a11
DevPayment.sol	21f29ca66c6907efdc8a1072905e956d

7.2 Re-Audit

Files	MD5 Hash
DevPayment	69c199d3b1f886803148c571a86f2326

8 Disclaimer

Shellboxes reports should not be construed as ”endorsements” or ”disapprovals” of particular teams or projects. These reports do not reflect the economics or value of any ”product” or ”asset” produced by any team or project that engages Shellboxes to do a security evaluation, nor should they be regarded as such. Shellboxes Reports do not provide any warranty or guarantee regarding the absolute bug-free nature of the examined technology, nor do they provide any indication of the technology’s proprietors, business model, business or legal compliance. Shellboxes Reports should not be used in any way to decide whether to invest in or take part in a certain project. These reports don’t offer any kind of investing advice and shouldn’t be used that way. Shellboxes Reports are the result of a thorough auditing process designed to assist our clients in improving the quality of their code while lowering the significant risk posed by blockchain technology. According to Shellboxes, each business and person is in charge of their own due diligence and ongoing security. Shellboxes does not guarantee the security or functionality of the technology we agree to research; instead, our purpose is to assist in limiting the attack vectors and the high degree of variation associated with using new and evolving technologies.

For a Contract Audit, contact us at contact@shellboxes.com