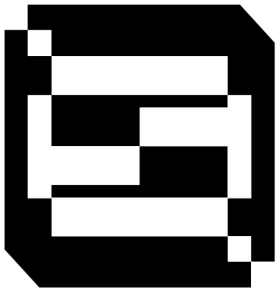

OFFSIDE Labs

# Mind SAP

**Smart Contract Security Assessment**

**January 2024**

**Prepared for:**

**Mind Network**


**Prepared by:**

**Offside Labs**

*Yao Li*

*Allen Xu*

*Siji Feng*

# Contents


**1 About Offside Labs** **2**

**2 Executive Summary** **3**

**3 Summary of Findings** **4**

**4 Key Findings and Recommendations** **5**

4.1 Reentrancy Vulnerability in _collectFee Function . . . . . 5
4.2 Unrestricted Access Control in BlackList Contract Functions . . . . . 6
4.3 Vulnerability to Signature Reuse and Delayed Execution in Signature Verification 7
4.4 Improper Nonce Management for Receiver in Service Account Transfers . . . . 8
4.5 Potential Loss of Funds due to Incorrect Token Transfer Handling and Fee Deduction Logic . . . . . 9
4.6 Lack of Zero Address Validation for Signature Recovery . . . . . 10
4.7 Lack of Constraints for Relayer Gas Fee . . . . . 11
4.8 Lack of Protocol Fee Deduction and Excess Payment Handling in _transferContractToSA . . . . . 12
4.9 User Experience Issues with Uncommon Fee Addition Mechanism . . . . . 13
4.10 Misleading Function Name and Incorrect Public Key Length Check in isPublicKeyExist . . . . . 14
4.11 Misuse of Nonce for Stealth Address Existence Check . . . . . 15
4.12 Lack of Sender Validation in _ccipReceive Function . . . . . 17
4.13 Informational and Undetermined Issues . . . . . 18

**5 Disclaimer** **21**

# 1 About Offside Labs

**Offside Labs** is a leading security research team, composed of top talented hackers from both academia and industry.

We possess a wide range of expertise in modern software systems, including, but not limited to, *browsers*, *operating systems*, *IoT devices*, and *hypervisors*. We are also at the forefront of innovative areas like *cryptocurrencies* and *blockchain technologies*. Among our notable accomplishments are remote jailbreaks of devices such as the **iPhone** and **PlayStation 4**, and addressing critical vulnerabilities in the **Tron Network**.

Our team actively engages with and contributes to the security community. Having won and also co-organized *DEFCON CTF*, the most famous CTF competition in the Web2 era, we also triumphed in the **Paradigm CTF 2023** within the Web3 space. In addition, our efforts in responsibly disclosing numerous vulnerabilities to leading tech companies, such as *Apple*, *Google*, and *Microsoft*, have protected digital assets valued at over **$300 million**.

In the transition towards Web3, Offside Labs has achieved remarkable success. We have earned over **$9 million** in bug bounties, and **three** of our innovative techniques were recognized among the **top 10 blockchain hacking techniques of 2022** by the Web3 security community.

https://offside.io/

https://github.com/offsidelabs

https://twitter.com/offside_labs

# 2 Executive Summary

### Introduction

*Offside Labs* completed a security audit of *Mind Network*'s *Mind SAP* smart contracts, starting on January 22, 2024, and concluding on January 25, 2024.

### Mind SAP Project Overview

*Mind Network* combines Fully Homomorphic Encryption with Stealth Address Protocol (FHE DK-SAP) to achieve private and secure value transfers, solving two problems simultaneously:

1. the need for privacy with Web3 data,
2. the security protocol necessary for Web2 and traditional finance to enter web3 while remaining compliant with regulators.

### Audit Scope

The assessment scope contains mainly the smart contracts and offchain utilities of the *Mind SAP* project for *Mind Network*.

The audit is based on the following specific branches and commit hashes of the codebase repositories:

- Mind SAP
  - Branch: master
  - Commit Hash: 922157bc02f1c0773bed46286f772414cb707cdd
  - Codebase Link

We listed the files we have audited below:

- Mind SAP
  - mind-sap-contract/contract/*
  - mind-sap-relay-wallet/src/*
  - mind-sap-sdk/src/*

### Findings

The security audit revealed:

- 1 critical issue
- 3 high issues
- 4 medium issues
- 4 low issues
- 4 informational issues

Further details, including the nature of these issues and recommendations for their remediation, are detailed in the subsequent sections of this report.

# 3 Summary of Findings

| ID | Title | Severity | Status |
|---|---|---|---|
| 01 | Reentrancy Vulnerability in _collectFee Function | Critical | Fixed |
| 02 | Unrestricted Access Control in BlackList Contract Functions | High | Fixed |
| 03 | Vulnerability to Signature Reuse and Delayed Execution in Signature Verification | High | Fixed |
| 04 | Improper Nonce Management for Receiver in Service Account Transfers | High | Fixed |
| 05 | Potential Loss of Funds due to Incorrect Token Transfer Handling and Fee Deduction Logic | Medium | Fixed |
| 06 | Lack of Zero Address Validation for Signature Recovery | Medium | Fixed |
| 07 | Lack of Constraints for Relayer Gas Fee | Medium | Fixed |
| 08 | Lack of Protocol Fee Deduction and Excess Payment Handling in _transferContractToSA | Medium | Fixed |
| 09 | User Experience Issues with Uncommon Fee Addition Mechanism | Low | Fixed |
| 10 | Misleading Function Name and Incorrect Public Key Length Check in isPublicKeyExist | Low | Fixed |
| 11 | Misuse of Nonce for Stealth Address Existence Check | Low | Fixed |
| 12 | Lack of Sender Validation in _ccipReceive Function | Low | Fixed |
| 13 | Suboptimal Placement of blackList.isProhibited() Check Leading to Unnecessary Gas Usage | Informational | Acknowledged |
| 14 | Integration of EIP-712 Standard for Structured Data Signing in _verifySASig | Informational | Acknowledged |
| 15 | Centralization Risk of SAPBridgeReceiver | Informational | Fixed |
| 16 | Absence of Upgradability and Pausability in Contract Design | Informational | Fixed |

# 4 Key Findings and Recommendations

## 4.1 Reentrancy Vulnerability in _collectFee Function

| Severity: Critical | Status: Fixed |
| --- | --- |
| Target: Smart Contract | Category: Reentrancy |

### Description

The smart contract contains a function `collectFee` that allows relayers to withdraw fees in the form of native cryptocurrency or ERC20 tokens. The internal function `_collectFee` is called to handle the logic of this withdrawal. The current implementation of the `_collectFee` function is vulnerable to reentrancy attacks because it updates the user's balance after making an external call to send funds, which can be exploited by a malicious actor.

### Impact

If exploited, a malicious actor could repeatedly withdraw funds during a single transaction before the `beneficiaryBalance` is set to zero. This could lead to a loss of funds far exceeding the intended withdrawal amount, potentially draining all funds in the contract.

### Proof of Concept

The vulnerability lies in the following lines of code within the `_collectFee` function:

```
(bool sent, ) = msg.sender.call{value: balance}("");
if (!sent) revert FailedToWithdrawEth(msg.sender, balance);
beneficiaryBalance[msg.sender][token] = 0;
```

Here's how the vulnerability could be exploited:

1. An attacker sets up a malicious contract as the `relayerWallet`.
2. The attacker intentionally generates fees, which are now associated with the `relayerWallet`'s address in the `beneficiaryBalance[]` mapping.
3. The attacker initiates a withdrawal via `collectFee`, which triggers the vulnerable `_collectFee` function.
4. The malicious `relayerWallet` contract, when receiving the native tokens, re-enters the `collectFee` function.
5. Since the `beneficiaryBalance` for the `relayerWallet` has not yet been set to zero due to the position of the state update after the external call, the re-entered `collectFee` function sees the balance as still available.
6. This process can be repeated as many times as the gas limit allows, draining all the funds in this protocol, including the deposited ETH of all users and fees of all beneficiaries.

This vulnerability could allow an attacker to steal not only their associated balance but potentially the balances of all other users, which could lead to a significant loss of funds.

#### Recommendation

To mitigate this issue, follow the checks-effects-interactions pattern by updating the user's balance before making the external call. The recommended change would be as follows:

```
function _collectFee(address token) internal {
    uint256 balance = beneficiaryBalance[msg.sender][token];
    if (balance > 0) {
        beneficiaryBalance[msg.sender][token] = 0;
        if (token == NATIVE_TOKEN) {
            (bool sent, ) = msg.sender.call{value: balance}("");
            if (!sent) revert FailedToWithdrawEth(msg.sender, balance);
        } else {
            SafeERC20.safeTransfer(IERC20(token), msg.sender, balance);
        }
    }
}
```

#### Mitigation Review Log

**Mind Network:** **Fixed** by setting state before external call in `_collectFee`. Commit 3c793590.

## 4.2 Unrestricted Access Control in BlackList Contract Functions

| Severity: High | Status: Fixed |
| --- | --- |
| Target: Smart Contract | Category: Access Control |

#### Description

The `BlackList` contract is designed to manage a list of addresses that are prohibited from performing certain actions. However, the `setProhibited` and `revokeProhibited` functions lack proper access control mechanisms, allowing any external entity to modify the blacklist.

#### Impact

Due to the absence of access control, any user or contract can arbitrarily add to or remove addresses from the blacklist, completely undermining the security and integrity of the blacklist functionality. This vulnerability renders the blacklist ineffective in restricting actions for prohibited addresses.

#### Recommendation

Using OpenZeppelin's Ownable contract to restrict these functions to only the contract owner.

#### Mitigation Review Log

**Mind Network:** **Fixed** by adding OpenZeppelin's Ownable and onlyOwner in `BlackList.sol` . Commit 2bcac332.

### 4.3 Vulnerability to Signature Reuse and Delayed Execution in Signature Verification

| Severity: High | Status: Fixed |
| --- | --- |
| Target: Smart Contract | Category: Signature |

#### Description

The current implementation of the `_verifySASig` function does not account for the possibility that a transaction could be reverted after a signature has been provided. This oversight allows for the potential extraction and reuse of the signature against the user's will. Furthermore, without a timestamp or expiration mechanism, a relayer (or any entity in possession of a valid signature) could deliberately withhold and execute the signed transaction at a later time when it is most advantageous to them, and potentially detrimental to the signer.

#### Impact

- **Unauthorized Reuse of Signatures:** If a transaction fails, the signature can be captured and reused maliciously, as it remains valid due to the nonce not being incremented on failed transactions.
- **Relayer Misconduct:** A relayer could exploit the lack of expiration by waiting to execute the transaction until the market conditions or contract state change in their favor.
- **User Autonomy Compromised:** Users may lose control over their signed transactions, as they cannot enforce execution within a specific timeframe or prevent execution after a certain period.

#### Recommendation

- **Nonce Increment on Reversion:** Ensure that nonces are incremented even when transactions are reverted to prevent signature reuse.
- **Timestamp or Expiration Check:** Introduce a timestamp or expiration time into the signature to prevent indefinite validity and execution outside of an acceptable timeframe.

- **Slashing Mechanism:** Implement a mechanism to penalize relayers who withhold transactions for personal gain.

#### Mitigation Review Log

**Mind Network:** **Fixed** by adding expire time in `RelayerRequest`. Commit 4e21e9ca.

### 4.4 Improper Nonce Management for Receiver in Service Account Transfers

| Severity: High | Status: Fixed |
| --- | --- |
| Target: Smart Contract | Category: Logic Error |

#### Description

Within the `_transferSAtoSA` function, the nonce is incorrectly incremented for the recipient `saDest`. Nonces are typically used to ensure transactions are processed in order and to prevent replay attacks. However, incrementing the nonce for the recipient deviates from the standard practice of nonce management.

#### Impact

This flaw could lead to a Denial of Service (DoS) condition for users. Attackers can transfer tokens with invalid or low value to a target service account, thereby unnecessarily incrementing the recipient's nonce. This could disrupt the recipient's ability to send out valid transactions due to mismatched nonce values.

#### Vulnerable Code

```
function _transferSAtoSA(address saSrc, address saDest, address token,
    uint224 amount, address payable relayerWallet, uint224 gas) internal {
    uint224 fee = _calcFee(THIS_CONTRACT, ACTION_SAtoSA, token, amount);
    uint224 total = amount + fee + gas;
    require(saAccount[saSrc][token].balance >= total, "Not enough token
        in SA balance");

    saAccount[saSrc][token].balance -= total;
    if (fee > 0) beneficiaryBalance[feeReceiver][token] += fee;
    saAccount[saDest][token].nonce += 1;
    saAccount[saDest][token].balance += amount;
    if (gas > 0) beneficiaryBalance[relayerWallet][token] += gas;
}
```

### Recommendation

Do not increase the nonce of the recipient.

### Mitigation Review Log

**Mind Network:** **Fixed** by removing nonce increasement for `dest account`. Commit 392c33c5.

## 4.5 Potential Loss of Funds due to Incorrect Token Transfer Handling and Fee Deduction Logic

| Severity: Medium | Status: Fixed |
| --- | --- |
| Target: Smart Contract | Category: Logic Error |

### Description

The smart contract assumes that the total amount of contract tokens received after calling `SafeERC20.safeTransferFrom` will be equal to the total value (amount plus fee). This does not account for non-standard ERC20 tokens that might charge a transfer fee, which can result in the contract receiving less than the expected total. This discrepancy could lead to accounting errors and potential loss of funds.

### Impact

If the received amount is less than the expected due to transfer fees charged by non-standard ERC20 tokens, the contract's balance accounting will be inaccurate. This can result in the contract believing it has more tokens than it actually does, potentially affecting subsequent transactions and leading to loss of funds for the users or the contract owner.

### Vunerable Code

```
function _transferContractToSA(address saDest, address token, uint224
    amount) internal {
    require(exemptList[msg.sender], "Not from a exempted contract");
    if (token == NATIVE_TOKEN) {
        /// Sending native token
        require(msg.value >= amount, "Not enough token sended to SA");
    } else {
        /// Sending contract token
        require(IERC20(token).balanceOf(msg.sender) >= amount, "Not
            enough token in wallet");
        SafeERC20.safeTransferFrom(IERC20(token), msg.sender,
    address(this), amount);
    }
    saAccount[saDest][token].nonce += 1;
    saAccount[saDest][token].balance += amount;
}
```

### Recommendation

Implement logic to calculate the actual amount received by the contract after the transfer. This can be done by checking the contract's balance before and after the transfer.

### Mitigation Review Log

**Mind Network:** **Fixed** by adding balance increase amount check in `_transferEOAtoSA`, `_transferContractToSA`, `_transferToEVMAddr`. Commit 7b5960cd.

## 4.6 Lack of Zero Address Validation for Signature Recovery

| Severity: Medium | Status: Fixed |
| --- | --- |
| Target: Smart Contract | Category: Signature |

### Description

The function `_verifySASig` is designed to recover an address from a signature and compare it with a source address `saSrc`. However, if the signature is invalid, the `ecrecover` function will return the zero address (0x0000000000000000000000000000000000000000). Currently, there is no check in place to verify that the recovered address is not the zero address, which could result in the function incorrectly passing the `require(addressRecover == saSrc)` statement if `saSrc` is also inadvertently set as the zero address. This could lead to confusion and unintended authorization, potentially allowing unauthorized actions to take place.

### Impact

The lack of validation against the zero address can lead to scenarios where an invalid signature could cause the function to behave incorrectly, especially if the `saSrc` parameter is not properly sanitized elsewhere in the contract. This could result in a security vulnerability where the contract incorrectly processes transactions as if they were signed by a legitimate address.

### Recommendation

Implement a check to ensure that the address recovered from the signature is not the zero address before proceeding with the nonce verification and any subsequent logic:

```
require(addressRecover != address(0), "Invalid signature");
```

### Mitigation Review Log

**Mind Network:** **Fixed** by adding zero address check in `_verifySASig` . Commit *ee60221d*.

## 4.7 Lack of Constraints for Relayer Gas Fee

| Severity: Medium | Status: Fixed |
| --- | --- |
| Target: Smart Contract | Category: Insufficient Validation |

### Description

User signs encoded message with merged gas fee and extra `SAPHandler` fee, and delegates it to relayer for avoidance of sending transaction directly. The encoded message is provided by relayer, but there's no constraint in smart contract to protect unreasonable fee.

### Impact

Malicious or misconfigured relayer may provide improper message data with high fee. If user doesn't check it carefully, associated tokens approved to `SAClientERC20` contract can be drained.

### Recommendation

Calculate approximate native token fee in smart contract, and convert to associated token amount by price oracle. Use it as cap to constrain relayer gas fee.

### Mitigation Review Log

**Mind Network:** **Fixed** by estimating gas at SDK client side.

**Offside Labs:**. The estimated gas from the SDK only works for NATIVE TOKEN. If the user requests a transfer of an ERC20 token, the value does not match. Commit 237d9025.

**Mind Network:** **Fixed** by supporting ERC20 token relayer fee. Commit ad7ac644.

## 4.8 Lack of Protocol Fee Deduction and Excess Payment Handling in _transferContractToSA

| Severity: Medium | Status: Fixed |
| --- | --- |
| Target: Smart Contract | Category: Insufficient Validation |

### Description

The `_transferContractToSA` function is responsible for transferring tokens from a contract to a Stealth Address (SA). It includes a check for exempt contracts and processes transfers for both native and ERC20 tokens. However, two significant issues are identified in the current implementation:

1. **Protocol Fees:** The function does not deduct any protocol fees when transferring to an SA. If this is an intentional design choice, it should be explicitly documented. Otherwise, it might be an oversight that could lead to inconsistent fee application across the platform.
2. **Excess Native Token Refund:** When transferring native tokens (e.g., ETH), the function requires that `msg.value` be at least equal to the amount to transfer. However, it does not account for scenarios where `msg.value` could be greater than amount. In such cases, the excess amount of native tokens sent is not refunded to the sender, which could lead to unintentional loss of funds for users.

### Impact

1. **Revenue Loss:** If exempting the transfer from protocol fees is not intentional, it could result in a loss of potential fee revenue.
2. **Balance Discrepancy:** The lack of excess payment refunds leads to an inconsistency between the internal accounting and the actual balance of the contract. This discrepancy complicates the financial tracking and auditing processes.

### Recommendation

1. **Protocol Fees:** Clearly document the intention behind not charging protocol fees for this transaction type or adjust the implementation to include fee deduction where appropriate.
2. **Excess Payment Refunds:** Modify the function to enforce that `msg.value` equals the amount being transferred, or implement a refund mechanism to return any excess payment to the sender.

### Mitigation Review Log

**Mind Network:** **Fixed** modifying condition check and adding comment in `_transferContractToSA` . Commit 1bc43c96.

## 4.9 User Experience Issues with Uncommon Fee Addition Mechanism

| Severity: Low | Status: Fixed |
| --- | --- |
| Target: Smart Contract & Type Script SDK | Category: Insufficient Validation |

### Description

The contract implements an atypical fee structure where the fee is added on top of the transferred amount (total = amount + fee), as opposed to being deducted from the transferred amount. This can create confusion and complicate the user experience as users have to calculate the total amount that needs to be approved for the contract, potentially leading to failed transactions or over-approvals.

### Impact

Users may not anticipate the extra fee and could end up approving an incorrect amount, leading to transaction failures. On the other hand, users providing maximal token allowance could unintentionally expose themselves to greater risk if the contract or its operator is compromised.

### Vulnerable Code

```
async approveERC20Token(approveAddress?: string) {
    const tokenContract = ERC20.connect(this.payload.token.address,
      ↪ this.signer);
    const amount = this.payload.amount as BigNumber;
    const address = await this.signer.getAddress();
    if (!approveAddress) {
        const chainId = await this.signer.getChainId();
        const chain = parseChainConfig(chainId);
        approveAddress = chain.ERC20ClientAddress;
    }
    const allowance = (await tokenContract.allowance(address,
      ↪ approveAddress)) as BigNumber;
    if (amount.gt(allowance)) {
        const approveTx = await tokenContract.approve(approveAddress,
          ↪ MaxUint256);
        await approveTx.wait();
    }
}
```

The client-side script compares the allowance to the desired sending amount, which is less than the actual total amount required. If more allowance is needed, it requests maximal allowance from the user.

### Recommendation

- Adjust the fee structure to be deducted from the transferred amount, aligning with common industry practices. This would mean the user sends a total of amount, and the fee is taken from this amount.
- Or, provide a utility function or view that calculates the total required approval amount based on the intended transfer amount, simplifying the user's approval process.

### Mitigation Review Log

**Mind Network:** **Fixed** by extra allowance check in sdk. Commit 0a017f50.

## 4.10 Misleading Function Name and Incorrect Public Key Length Check in isPublicKeyExist

| Severity: Low | Status: Fixed |
| --- | --- |
| Target: Type Script SDK | Category: Design Issue |

#### Description

The function is intended to validate public keys and ensure they are not zero addresses. However, the function name implies a check for the existence of a public key rather than its validity. Given that the key will be used in `KeypairECC` construction, it would be more appropriate to assert that the length of the pubkey corresponds to that of a public key (68 characters), rather than to that of a private key (66 characters).

#### Impact

This could lead to runtime errors or security issues if invalid keys are not properly identified. For example, a private key could pass the check and be used to construct a `KeyPairECC` .

#### Vulnerable Code

```
export async function createSA(receipt: string) {
    const keys = await getKeys(receipt);
    if (!isPublicKeyExist(keys.opPubKey)) {
        throw new NO_REGISTER();
    }
    const opKeypair = new KeypairECC({ key: keys.opPubKey });
```

#### Recommendation

- Assert that the length is 68 characters to validate a public key.
- There is no need to construct a BigNumber with a padded hex string to represent zero.

#### Mitigation Review Log

**Mind Network:** **Fixed** in Commit a1fe70e9.

**Offside Labs:** The major concern with this issue is the ambiguity about the use of the class `KeyPairECC` . The constructor of KeyPair accepts both a private key and a public key, which are distinguished by the key.length. It is recommended to have an additional parameter indicating the type of key being used.

**Mind Network:** **Fixed** by adding additional parameter for `KeyPairECC` . Commit 0af38307.

### 4.11 Misuse of Nonce for Stealth Address Existence Check

| Severity: Low | Status: Fixed |
| --- | --- |
| Target: Smart Contract | Category: Design Issue |

### Description

The `existSA` function is intended to verify the existence of a Stealth Address (SA) based on non-zero nonce values associated with a list of specified tokens. However, this method is flawed for two reasons:

1. Incrementing the nonce for an SA as a receiver can lead to false positives. If an SA has received a token transaction that is not included in the tokens parameter, its nonce would be incremented, suggesting its existence even if it has not engaged in any outgoing transactions.
2. The reliance on the nonce as an existence indicator is problematic. Nonces should be reserved for tracking the sequence of outgoing transactions to prevent replay attacks, not for confirming the existence of an SA.

### Impact

The current logic could produce inaccurate results, potentially causing users or dependent applications to misunderstand the status and transaction history of an SA. This could lead to oversight of asset holdings, incorrect balance reporting, and vulnerabilities in transaction handling.

### Vulnerable Code

```
function existSA(address sa, address[] calldata tokens) external view
    returns (bool isExist) {
    for (uint256 i = 0; i < tokens.length; i++) {
        address token = tokens[i];
        if (saAccount[sa][token].nonce > 0) return true;
    }
    return false;
}
```

### Recommendation

The `existSA` function checks the mapping to determine if the SA has been initialized. Nonces can now solely be used for their intended purpose—tracking the number of outgoing transactions to prevent replay attacks—without conflating with existence verification.

```
/// Mapping to track the existence of Stealth Addresses
    mapping(address => bool) private saExists;
```

### Mitigation Review Log

**Mind Network:** **Fixed** by adding bool exist field for `Account`. Commit 392c33c5.

## 4.12 Lack of Sender Validation in _ccipReceive Function

| Severity: Low | Status: Fixed |
| --- | --- |
| Target: Smart Contract | Category: Insufficient Validation |

### Description

The `_ccipReceive` function is currently missing a crucial validation step to confirm that the `message.sender` is, in fact, the expected sender, specifically the `SapBridge` from the other chain.

### Impact

The absence of this validation allows the function to potentially accept messages from unauthorized sources. This could result in security vulnerabilities, including the execution of unauthorized actions or undesired state alterations within the contract.

### Vulnerable Code

```
function _ccipReceive(
    Client.Any2EVMMessage memory message
) internal override {
    bytes32 latestMessageId = message.messageId;
    uint64 latestSourceChainSelector = message.sourceChainSelector;
    address latestSender = abi.decode(message.sender, (address));
    (address sa, bytes memory ciphertext) = abi.decode(
        message.data, (address, bytes)
    );
```

### Recommendation

It is recommended to implement a whitelist for message.sender, mandating that the message.sender must be the exact address of the `SapBridge` .

Additionally, there are some best practices outlined at https://docs.chain.link/ccip/best-practices, such as verifying the source/destination chain ID. Follow these guidelines as necessary.

### Mitigation Review Log

**Mind Network:** **Fixed** by adding whitelist sender list in `_ccipReceive` . Commit 86a143ce.

## 4.13 Informational and Undetermined Issues

### Suboptimal Placement of blackList.isProhibited() Check Leading to Unnecessary Gas Usage

| Severity: Informational | Status: Acknowledged |
| --- | --- |
| Target: Smart Contract | Category: Gas Optimization |

The smart contract functions that utilize the `blackList.isProhibited()` check are currently invoking this validation step at the end of the function logic. This placement is inefficient because it allows for the execution of potentially gas-intensive operations before determining if the interacting wallet address is prohibited. If the address is indeed blacklisted, the entire transaction reverts after having expended unnecessary gas, resulting in a waste of resources for the user.

Users interact with functions that may execute multiple operations consuming gas, only to fail at the final step if their address is blacklisted. This not only wastes gas but also degrades the user experience, as transactions that will inevitably fail are allowed to execute significant portions of their logic.

- Relocate the `blackList.isProhibited()` check to the beginning of the respective functions. This ensures that no gas is expended on operations within a transaction that is destined to fail due to a blacklisted address.
- Implement a modifier that encapsulates the blacklist check and apply it to all functions that require this validation. This establishes a consistent pattern and makes the code more maintainable.

**Mind Network: Acknowledged.** Intentional design

### Integration of EIP-712 Standard for Structured Data Signing in _verifySASig

| Severity: Informational | Status: Acknowledged |
| --- | --- |
| Target: Smart Contract | Category: Signature |

The `_verifySASig` function uses `ECDSA` signatures without the benefits of structured data. `EIP-712` can enhance this by enabling human-readable and securely signed transactions, improving both user experience and security.

Without `EIP-712` , users may not fully understand what they are signing, potentially leading to security risks. Adoption of EIP-712 would align `_verifySASig` with industry best practices.

Adopt `EIP-712` in `_verifySASig` to use structured data for signing, making transactions more transparent and secure for users.

- EIP-712 Specification: Ethereum EIPs - EIP-712
- EIP-712 Improvement Proposal: Ethereum GitHub

**Mind Network: Acknowledged.** The signing process of SA private key won't trigger wallet extension, so human readability isn't necessary.

### Centralization Risk of SAPBridgeReceiver

| Severity: Informational | Status: Fixed |
| --- | --- |
| Target: Smart Contract | Category: Centralization Risk |

The `SAPBridgeReceiver` contract contains functions that allow the owner to set a client contract for ERC20 tokens ( `setSAClientERC20` ) and to approve an unlimited amount of a token to be spent by this client contract ( `approveToken` ). This design introduces a high level of centralization.

If the owner's private key were compromised, an attacker could redirect incoming funds to any address by updating the `_saClientERC20` address and draining the approved ERC20 tokens. Users must trust the owner not to act maliciously.

```
function setSAClientERC20(address saClientERC20) public onlyOwner {
    _saClientERC20 = saClientERC20;
}

function approveToken(address token) public onlyOwner {
    IERC20(token).approve(_saClientERC20, type(uint256).max);
}
```

**Mind Network:** **Fixed** by removing `setSAClientERC20` and making `saClientERC20` immutable.

**Mind Network:** **Fixed** by setting `_saClientERC20` immutable in `SAPBridgeReceiver.sol` . Commit d80562ed.

### Absence of Upgradability and Pausability in Contract Design

| Severity: Informational | Status: Fixed |
| --- | --- |
| Target: Smart Contract | Category: Design Issue |

The current contract architecture lacks upgradability and pausability design patterns. Upgradability allows for the modification and enhancement of the contract's logic post-deployment, while pausability enables the contract to halt operations in the case of an emergency or to address potential issues.

Recommendation:

- Implement Upgradable Design
- Introduce Pausability

References:

- OpenZeppelin Upgrades
- OpenZeppelin Pausable

- Diamond Standard EIP-2535

  **Mind Network:** **Fixed** by adding OpenZeppelin's Upgradable Proxy and Pausable in `SAClientERC20.sol` . Commit 22068f7c.

# 5 Disclaimer

This audit report is provided for informational purposes only and is not intended to be used as investment advice. While we strive to thoroughly review and analyze the smart contracts in question, we must clarify that our services do not encompass an exhaustive security examination. Our audit aims to identify potential security vulnerabilities to the best of our ability, but it does not serve as a guarantee that the smart contracts are completely free from security risks.

We expressly disclaim any liability for any losses or damages arising from the use of this report or from any security breaches that may occur in the future. We also recommend that our clients engage in multiple independent audits and establish a public bug bounty program as additional measures to bolster the security of their smart contracts.

It is important to note that the scope of our audit is limited to the areas outlined within our engagement and does not include every possible risk or vulnerability. Continuous security practices, including regular audits and monitoring, are essential for maintaining the security of smart contracts over time.

By using this report, the client acknowledges the inherent limitations of the audit process and agrees that our firm shall not be held liable for any incidents that may occur subsequent to our engagement.

This report is considered null and void if the report (or any portion thereof) is altered in any manner.