verichains

SECURITY AUDIT OF

FCO SMART CONTRACT

Public Report

Dec 20, 2023

Verichains Lab

info@verichains.io

https://www.verichains.io

Driving Technology > Forward

ABBREVIATIONS

Name	Description
Ethereum	An open source platform based on blockchain technology to create and distribute smart contracts and decentralized applications.
Ether (ETH)	A cryptocurrency whose blockchain is generated by the Ethereum platform. Ether is used for payment of transactions and computing services in the Ethereum network.
Smart contract	A computer protocol intended to digitally facilitate, verify or enforce the negotiation or performance of a contract.
Solidity	A contract-oriented, high-level language for implementing smart contracts for the Ethereum platform.
Solc	A compiler for Solidity.
ERC20	ERC20 (BEP20 in Binance Smart Chain or *x*RP20 in other chains) tokens are blockchain-based assets that have value and can be sent and received. The primary difference with the primary coin is that instead of running on their own blockchain, ERC20 tokens are issued on a network that supports smart contracts such as Ethereum or Binance Smart Chain.

EXECUTIVE SUMMARY

This Security Audit Report was prepared by Verichains Lab on Dec 20, 2023. We would like to thank the FCO for trusting Verichains Lab in auditing smart contracts. Delivering high-quality audits is always our top priority.

This audit focused on identifying security flaws in code and the design of the FCO Smart Contract. The scope of the audit is limited to the source code files provided to Verichains. Verichains Lab completed the assessment using manual, static, and dynamic analysis techniques.

During the audit process, the audit team had identified some vulnerability issues in the contract code.

TABLE OF CONTENTS

1. MANAGEMENT SUMMARY

1.1. About FCO Smart Contract

Fanatico offers several innovative apps to fans for harnessing the full potential of the FCO token, the heart of its blockchain-powered ecosystem.

1.2. Audit scope

This audit focused on identifying security flaws in code and the design of the FCO Smart Contract.

The audited contract is the FCO Smart Contract that deployed on Polygon Mainnet at address `0xc058228fdc4c4d70fdd324bdf8377e3c64564450` (behind proxy at address `0x406B9D726F47F9928Ec794beB2cE12B251D13852`). The details of the deployed smart contract are listed in Table 1.

FIELD	VALUE
Contract Name	FCOToken
Contract Implement Address	0xc058228fdc4c4d70fdd324bdf8377e3c64564450
Contract Proxy Address	0x406B9D726F47F9928Ec794beB2cE12B251D13852
Compiler Version	v0.8.19+commit.7dd6d404
Optimization Enabled	Yes with 200 runs
Explorer	*https://polygonscan.com/address/0xc058228fdc4c4d70fdd324bdf8377e3c64564450*

Table 1. The deployed smart contract details

1.3. Audit methodology

Our security audit process includes four steps:

- Mechanism Design is reviewed to look for any potential problems.
- Source codes are scanned/tested for commonly known and more specific vulnerabilities using public and our in-house security analysis tool.
- Manual audit of the codes for security issues. The source code is manually analyzed to look for any potential problems.
- Set up a testing environment to debug/analyze found issues and verifies our attack PoCs.

For vulnerabilities, we categorize the findings into categories as listed in table below, depending on their severity level:

SEVERITY LEVEL	DESCRIPTION
CRITICAL	A vulnerability that can disrupt the functioning; creates a critical risk to the application; required to be fixed immediately.
HIGH	A vulnerability that could affect the desired outcome of executing the application with high impact; needs to be fixed with high priority.
MEDIUM	A vulnerability that could affect the desired outcome of executing the application with medium impact in a specific scenario; needs to be fixed.
LOW	An issue that does not have a significant impact, can be considered as less important.

Table 2. Severity levels

1.4. Disclaimer

FCO acknowledges that the security services provided by Verichains, are conducted to the best of their professional abilities but cannot guarantee 100% coverage of all security vulnerabilities. FCO understands and accepts that despite rigorous auditing, certain vulnerabilities may remain undetected. Therefore, FCO agrees that Verichains shall not be held responsible or liable, and shall not be charged for any hacking incidents that occur due to security vulnerabilities not identified during the audit process.

1.5. Acceptance Minute

This final report served by Verichains to the FCO will be considered an Acceptance Minute. Within 7 days, if no any further responses or reports is received from the FCO, the final report will be considered fully accepted by the FCO without the signature.

2. AUDIT RESULT

2.1. Overview

The FCO Smart Contract was written in `Solidity` language, with the required version to be `^0.8.19`. The source code was written based on OpenZeppelin`s library.

2.1.1. AccessControl.sol

The `Authority` contract extends the contracts `Initializable`, `IAuthority`, and `AccessControl`. The `Authority` contract implements access control mechanisms based on roles, with `admin` and `operators` being the two main roles. The address of the `Authority` contract will be used by the `AccessControl` contract to determine the addresses of these two roles. Changing the address of the `Authority` contract may result in changes to the addresses associated with the `admin` and `operators` roles. Furthermore, the `AccessControl` contract defines a `recover()` function that can be used to transfer any tokens in the contract to any arbitrary address.

2.1.2. EventEmitter.sol

The `EventEmitterHub` contract extends the `AccessControl` contract. The `AccessControl` contract implements access control mechanisms based on roles. The admin role is capable of modifying the list of `emitters`, and the `emitters` are allowed to emit the `Event()` event.

2.1.3. FCOToken.sol

The `FCOToken` contract extends the contracts `ERC20Upgradeable`, `ERC20BurnableUpgradeable`, `ERC20FlashMintUpgradeable`, `AccessControl` and `EventEmitter`. `ERC20BurnableUpgradeable` is an extension of the ERC20 standard that adds the ability to burn tokens. This is useful for cases where the token needs to be burned, such as when a token is being upgraded. `ERC20FlashMintUpgradeable` is a contract that provides the ability to flash mint tokens. This is useful for cases where a contract needs to mint tokens in order to perform an action, such as when a contract is being upgraded.

The main functions in the contract are as follows:

- Admin can change the values of the variables `signUpReward` and `visitReward`.
- Operators can use the `mint()` and `mintBatch()` functions to mint tokens to specific addresses, the `lock()` function to lock additional tokens for users, and the `use()` function to use both locked tokens and tokens available in a user's wallet (available tokens will be burned, and locked tokens will be unlocked early).
- The `processRewards()` function allows operators to process rewards for any address. To use this function, ordinary users must have the operator's signature. The user will receive the `signUpReward` for the first time and the `visitReward` for subsequent times.

- Tokens will be automatically unlocked when they reach their expiration date and minted to the user when they transfer tokens.

The smart contract is `ERC20` implementation that have some properties (as of the report writing time):

PROPERTY	VALUE
Name	Fanatico
Symbol	FCO
Decimals	18
Total Supply	Undefined

Table 3. The FCO Smart Contract properties

For the ERC20 token, the security audit team has the list of centralization issues below:

Checklist	Status	Passed
Upgradeable	Yes	
Fee modifiable	No	Yes
Mintable	Yes	
Burnable	Yes	
Pausable	No	Yes
Trading cooldown	No	Yes
Has blacklist	No	Yes
Has whitelist	No	Yes

Table 4. The decentralization checklist

2.1.4. PublicationHub.sol

The `PublicationHub` contract extends the contracts `ERC1155Upgradeable`, `AccessControl` and `EventEmitter`.

When the contract is initialized, the native token and FCO token will be added to the `paymentTokens` list. The admin has the ability to change the `serviceWallet`, `signerWallet`, and `paymentTokens` at any time.

The main functions in the contract are as follows:

- `mint()` function: Any user can own a publication by paying a fee with the signature of `signerWallet` and the author's signature of the publication. Furthermore, based on `signerWallet`'s signature, the user can own an `encryptedId` or `decryptedId` item.
- The owner can transfer ownership to anyone using the `transferOwner()` function, along with the `encryptedId` (if available).
- The owner can also burn an `encryptedId` item to mint a `decryptedId` item using the `decrypt()` function.
- Any user can become a new owner or own a `decryptedId` item in the `collect()` function as long as they have the signatures of `signerWallet` and the owner and pay a fee. They also have the option to decrypt the item if they so desire.
- An address can only have one `encryptedId` item and/or one `decryptedId` item.
- When a user transfers an `encryptedId` item, ownership is transferred to the recipient.

2.2. Findings

During the audit process, the audit team found some vulnerabilities in the given version of FCO Smart Contract.

Severity	Name	Status
HIGH	Cross-chain signature replay	FIXED
HIGH	Use `tx.origin` for authentication is vulnerable to phishing	FIXED
MEDIUM	Users will lose money if they unlock and then lock within the same epoch	FIXED
MEDIUM	No Storage Gap for Upgradeable Contract Might Lead to Storage Slot Collision	FIXED
LOW	Missing logic to check `msg.value` in the `mint()` and `collect()` functions	FIXED

Severity	Name	Status
LOW	Users risk losing money if they deposit an ERC20 token but unintentionally attach a native token	FIXED
LOW	Admin may add a duplicated address in `setPaymentToken()` function	FIXED
LOW	Consider adding the `cancelCollect()` function	ACKNOWLEDGED
LOW	`epochsState.unlocked` has not been updated in the `_unlock()` function	ACKNOWLEDGED
INFORMATIVE	Should have events for `setServiceWallet()` and `setSignerWallet()` functions	FIXED
INFORMATIVE	Excessive variables and logic	FIXED
INFORMATIVE	Best practices to defend against reentrancy attacks	ACKNOWLEDGED
INFORMATIVE	Avoid relying on tx.origin for authentication	ACKNOWLEDGED

2.2.1. Cross-chain signature replay HIGH

Positions:

- `FCOToken.sol`#processRewards()
- `PublicationHub.sol`#mint()
- `PublicationHub.sol`#collect()

Description:

Cross-chain replay attacks arise when signatures can be reused across different blockchain systems. Once a signature has been used and invalidated on one chain, an attacker can still copy it, use it on another, and trigger an unwanted state change. This poses a significant threat to smart contract systems deployed across chains with identical code.

Impact:

- Users can copy the signature used for the `mint()` and `collect()` functions and use them on other chains, even if those chains do not initially allow it.

- Users who have an address's signature as an operator can re-use it on other chains to receive rewards via the `processRewards()` function as long as that address remains an operator on those chains.

```
function processRewards(RewardsData calldata rewardsData) public returns (RewardsResults[]
memory results) {
    uint256 length = rewardsData.rewards.length;
    if (length == 0) return results;

    if (!authority.operators(msg.sender)) {
        if (rewardsData.rewards[0].epochs.length == 0) return results;
        address signer =
ECDSAUpgradeable.recover(ECDSAUpgradeable.toEthSignedMessageHash(keccak256(abi.encode(rewar
dsData.rewards))), rewardsData.signature);
        require(authority.operators(signer), "Bad rewards signature");
    }

    ...
}
```

RECOMMENDATION

To mitigate this risk, the chain ID should be encoded in the signature payload.

UPDATES

- *Nov 10, 2023*: This issue has been acknowledged and fixed by the FCO team.

2.2.2. Use `tx.origin` for authentication is vulnerable to phishing HIGH

Position:

- `EventEmitter.sol`#L32

Description:

The smart contract uses `tx.origin` for delegation, making it susceptible to phishing attacks where a malicious contract can deceive an `admin` to execute a function and call the `register()` function inside the `EventEmitterHub` contract, setting the attacker's contract as a valid `emitters`.

```
function register() public {
    require(tx.origin == authority.admin(), "Not allowed");
    emitters[msg.sender] = true;
}
```

RECOMMENDATION

The `msg.sender` should be used instead of `tx.origin` for authentication purposes. For example, only the admin should be able to call this function:

```
function register() public onlyAdmin {
    emitters[msg.sender] = true;
}
```

UPDATES

- *Nov 03, 2023*: This issue has been acknowledged and fixed by the FCO team.

2.2.3. Users will lose money if they unlock and then lock within the same epoch MEDIUM

Positions:

- FCOToken.sol

Process:

Condition occurs: lockDuration < epochDuration, and the user is locked, unlocked, and locked again within the same epoch.

- The user is locked with 100 tokens in epoch 6 (epochDuration = 3s, lockDuration = 1s, and the timestamps of the epochs are sequentially 3s - 6s - 9s - 12s...).
  - epoch[6].locked = 100 tokens
  - epoch[6].timestamp = 6
  - epochsState.unlocked = 3
  - epochsState.first = 6
- Next, the user transfers tokens within epoch 6 (at 7s), and the _unlock(amount = 0) function is called. epochsState.unlocked is set to the current epoch's timestamp.
  - epochsState.unlocked = epoch[6].timestamp = 6
- If the user is locked again with an additional 200 tokens in epoch 6 (at 8s) by lock() function, the code at lines 312-315 does not execute anymore because epochsState.first is not 0. As a result, epochsState.unlocked remains at 6.
  - epoch[6].locked = 300 tokens
  - epoch[6].unlocked = 100 tokens
  - epochsState.unlocked = 6
  - epochsState.first = 6
- Finally, when the user calls _unlock(), the epoch starts from epoch 9 (line 328), and the second lock in epoch 6 is skipped, resulting in a loss of 200 tokens.

```
function lock(address account, uint256 amount) public onlyOperator {
    _nonZeroAmount(amount, true);
    _lock(account, amount);
}

function _lock(address account, uint256 amount) private {
    EpochsState storage epochsState = epochsStates[account];
    uint40 currEpoch = currentEpoch();
```

```
    Epoch storage epoch = epochs[account][currEpoch];

    epoch.locked += amount;

    if (epoch.timestamp == 0) {
        epoch.timestamp = currEpoch;
    }

    if (epochsState.first == 0) {  //<-- The first time you get locked, if you get locked a
second time, it won't satisfy this condition.
        epochsState.first = currEpoch;
        epochsState.unlocked = currEpoch - epochDuration;
    }

    if (epochsState.last < currEpoch) {
        epochsState.last = currEpoch;
    }

    emitEvent("FCO_LOCK", abi.encode(account, amount));
}

function _beforeTokenTransfer(address from, address to, uint256 amount) internal override {
    _nonZeroAmount(amount, true);

    if (from != address(0)) {
        uint256 unlocked = _unlock(from, 0); // unlock all possible locks first if they
expired in every transfer    <-- _unlock(amount = 0) is called
        if (unlocked != 0) _mint(from, unlocked);
    }
    ...
}

function _unlock(address account, uint256 amount) private returns (uint256 unlocked) {
    EpochsState storage epochsState = epochsStates[account];
    if (epochsState.first == 0 || epochsState.unlocked == epochsState.last) return
unlocked; // if no locks present skip next

    for (uint256 epochTimestamp = epochsState.unlocked; epochTimestamp < epochsState.last;)
{
        epochTimestamp += epochDuration;
        Epoch storage epoch = epochs[account][epochTimestamp];

        if (epoch.locked == 0) continue;

        if (amount == 0) { // if unlock amount not specified
            if (epoch.timestamp + lockDuration <= block.timestamp) { // unlock all expired
locks
                unlocked += epoch.locked - epoch.unlocked;  // unlock entire
```

```
                epoch.unlocked = epoch.locked;
                epochsState.unlocked = uint40(epoch.timestamp);  // shift unlocked forward
// <-- epochsState.unlocked receives a new timestamp value instead of using (timestamp -
epochDuration) as in the first lock
            } else {
                break;
            }
        } else { // if unlock amount requested (consumer)
            ...
        }
    }
    ...
}
```

UPDATES

- *Nov 03, 2023*: This issue has been acknowledged and fixed by the FCO team.

2.2.4. No Storage Gap for Upgradeable Contract Might Lead to Storage Slot Collision MEDIUM

Positions:

- AccessControl.sol
- EventEmitter.sol
- FCOToken.sol
- PublicationHub.sol

Description:

For upgradeable contracts, there must be storage gap to "allow developers to freely add new state variables in the future without compromising the storage compatibility with existing deployments". Otherwise, it may be very difficult to write new implementation code. Without storage gap, the variable in child contract might be overwritten by the upgraded base contract if new variables are added to the base contract. This could have unintended and very serious consequences of the child contracts.

Refer to the bottom part of this article: *https://docs.openzeppelin.com/upgrades-plugins/1.x/writing-upgradeable*

RECOMMENDATION

Recommend adding appropriate storage gap at the end of upgradeable contracts such as the below. Please reference OpenZeppelin upgradeable contract templates.

```
uint256[50] private __gap;
```

UPDATES

- *Nov 03, 2023*: This issue has been acknowledged and fixed by the FCO team.

2.2.5. Missing logic to check `msg.value` in the `mint()` and `collect()` functions LOW

Position:

- `PublicationHub.sol`#L147
- `PublicationHub.sol`#L291

Description:

When users choose to pay with native tokens, there is no check to verify how many native tokens they provide or if it matches the specified `price`. If there are native tokens available in the contract, attackers can potentially use them for distribution.

```
function mint(
    MintData calldata mintData_,
    uint256 tokenId_,
    uint256 deadline_,
    bytes calldata authorSignature_,
    bytes calldata serviceSignature_,
    IFCOToken.ApproveWithSignData calldata approveWithSignData,
    IFCOToken.RewardsData calldata processRewardsData
    ) payable public {
    ...

    uint256 encryptedId = getEncryptedId(tokenId_);

    Publication storage publication = publications[encryptedId];

    ...

    uint256 price = mintData_.price;
    address paymentToken = mintData_.paymentToken;

    _payout(publication, paymentToken, price, approveWithSignData, processRewardsData,
true);
}

    function _payout(
        Publication memory publication,
        address paymentTokenAddress,
        uint256 price,
        ...
    ) internal {
        PaymentToken memory paymentToken = paymentTokens[paymentTokenAddress];
        require(paymentToken.enabled, "Payment token not supported");
```

```
        require(price >= feeBase, "Price too low");

        ...

        if (minting) {
            authorAmount = price - serviceAmount;
        } else {
            ...
        }
        // missing check for msg.value
        if (paymentTokenAddress == address(0)) {
            if (authorAmount != 0) {
                (bool authorSuccess, ) = payable(publication.author).call{value:
authorAmount }("");
                require(authorSuccess, "Author payment error");
            }
            if (ownerAmount != 0) {
                (bool ownerSuccess, ) = payable(publication.owner).call{ value: ownerAmount
}("");
                require(ownerSuccess, "Owner payment error");
            }
            (bool serviceSuccess, ) = payable(serviceWallet).call{value: serviceAmount
}("");
            require(serviceSuccess, "Service payment error");
        } else {
            if (paymentTokenAddress == address(fco)) {
                require(approveWithSignData.data.amount == price, "Approve FCO wrong
amount");
                ...
            }
            if (authorAmount != 0) {
                IERC20Upgradeable(paymentTokenAddress).safeTransferFrom(msg.sender,
publication.author, authorAmount);
            }
            if (ownerAmount != 0) {
                IERC20Upgradeable(paymentTokenAddress).safeTransferFrom(msg.sender,
publication.owner, ownerAmount);
            }
            IERC20Upgradeable(paymentTokenAddress).safeTransferFrom(msg.sender,
serviceWallet, serviceAmount);
        }
    }
```

RECOMMENDATION

Should add a `require()` statement to compare `price` with `msg.value`:

```
function _payout(
        Publication memory publication,
        address paymentTokenAddress,
        uint256 price,
```

```
        ...
    ) internal {
        PaymentToken memory paymentToken = paymentTokens[paymentTokenAddress];
        require(paymentToken.enabled, "Payment token not supported");
        require(price >= feeBase, "Price too low");

        ...

        if (paymentTokenAddress == address(0)) {
            require(price == msg.value, "wrong native token amount");
            if (authorAmount != 0) {
                (bool authorSuccess, ) = payable(publication.author).call{value:
authorAmount }("");
                require(authorSuccess, "Author payment error");
            }
            if (ownerAmount != 0) {
                (bool ownerSuccess, ) = payable(publication.owner).call{ value: ownerAmount
}("");
                require(ownerSuccess, "Owner payment error");
            }
            (bool serviceSuccess, ) = payable(serviceWallet).call{value: serviceAmount
}("");
            require(serviceSuccess, "Service payment error");
        } else {
            ...
        }
    }
```

UPDATES

- *Nov 03, 2023*: This issue has been acknowledged and fixed by the FCO team.

2.2.6. Users risk losing money if they deposit an ERC20 token but unintentionally attach a native token LOW

Position:

- `PublicationHub.sol`#L147
- `PublicationHub.sol`#L291

Description:

The `mint()` and `collect()` functions allow users to deposit their tokens into the dApp. However, the function will not fail if the token is an ERC20, but the user attaches native token when calling this function. This is because the function is a payable function. As a result, the user will lose their money.

```
function mint(
    MintData calldata mintData_,
```

```
    uint256 tokenId_,
    uint256 deadline_,
    bytes calldata authorSignature_,
    bytes calldata serviceSignature_,
    IFCOToken.ApproveWithSignData calldata approveWithSignData,
    IFCOToken.RewardsData calldata processRewardsData
    ) payable public {
    ...

    _payout(publication, paymentToken, price, approveWithSignData, processRewardsData,
true);
}

    function _payout(
        Publication memory publication,
        address paymentTokenAddress,
        uint256 price,
        ...
    ) internal {
        ...
        if (paymentTokenAddress == address(0)) {
            ...
        } else {
            // missing check for msg.value == 0
            if (paymentTokenAddress == address(fco)) {
                require(approveWithSignData.data.amount == price, "Approve FCO wrong
amount");
                ...
            }
            if (authorAmount != 0) {
                IERC20Upgradeable(paymentTokenAddress).safeTransferFrom(msg.sender,
publication.author, authorAmount);
            }
            if (ownerAmount != 0) {
                IERC20Upgradeable(paymentTokenAddress).safeTransferFrom(msg.sender,
publication.owner, ownerAmount);
            }
            IERC20Upgradeable(paymentTokenAddress).safeTransferFrom(msg.sender,
serviceWallet, serviceAmount);
        }
    }
```

RECOMMENDATION

We recommend adding a requirement to check if the user attaches ETH or not when the token is an ERC20 by checking the `msg.value == 0`.

```
function _payout(
    Publication memory publication,
    address paymentTokenAddress,
```

```
    uint256 price,
    ...
) internal {
    ...
    if (paymentTokenAddress == address(0)) {
        ...
    } else {
        require(msg.value == 0, "Not accept native token");
        if (paymentTokenAddress == address(fco)) {
            require(approveWithSignData.data.amount == price, "Approve FCO wrong amount");
            ...
        }
        if (authorAmount != 0) {
            IERC20Upgradeable(paymentTokenAddress).safeTransferFrom(msg.sender,
publication.author, authorAmount);
        }
        if (ownerAmount != 0) {
            IERC20Upgradeable(paymentTokenAddress).safeTransferFrom(msg.sender,
publication.owner, ownerAmount);
        }
        IERC20Upgradeable(paymentTokenAddress).safeTransferFrom(msg.sender, serviceWallet,
serviceAmount);
    }
}
```

UPDATES

- *Nov 03, 2023*: This issue has been acknowledged and fixed by the FCO team.

2.2.7. Admin may add a duplicated address in `setPaymentToken()` function LOW

Position:

- `PublicationHub.sol`#L124

Description:

The `setPaymentToken()` function does not verify whether the `address_` passed into it exists or not. As a result, the admin can add a duplicated address and modify the data for that address within `paymentTokens`.

```
function setPaymentToken(address addess_, PaymentToken memory paymentToken_) public
onlyAdmin {
    paymentTokensList.push(addess_);
    paymentTokens[addess_] = paymentToken_;
    emitEvent("HUB_PAYMENT_TOKEN", abi.encode(addess_, paymentToken_));
}
```

RECOMMENDATION

Should have a `require` statement to check for duplicates when setting a payment token, and consider defining an `updatePaymentToken` function to prevent confusion

UPDATES

- *Nov 03, 2023*: This issue has been acknowledged and fixed by the FCO team.

2.2.8. Consider adding the `cancelCollect()` function LOW

Position:

- `PublicationHub.sol`

Description:

Should also consider adding the `cancelCollect()` function to allow the owner to change the price or stop `collect()` by blacklisting the old signature (add a nonce field to the `CollectData` struct and blacklist the message hash).

UPDATES

- *Nov 03, 2023*: This issue has been acknowledged by the FCO team.

2.2.9. `epochsState.unlocked` has not been updated in the `_unlock()` function LOW

Positions:

- `FCOToken.sol`#L323

Description:

When an epoch is completely unlocked, the value of the `epochsState.unlocked` variable should be updated.

```
function _unlock(address account, uint256 amount) private returns (uint256 unlocked) {
    EpochsState storage epochsState = epochsStates[account];
    if (epochsState.first == 0 || epochsState.unlocked == epochsState.last) return
unlocked; // if no locks present skip next

    for (uint256 epochTimestamp = epochsState.unlocked; epochTimestamp < epochsState.last;)
{
        epochTimestamp += epochDuration;
        Epoch storage epoch = epochs[account][epochTimestamp];

        if (epoch.locked == 0) continue;

        if (amount == 0) { // if unlock amount not specified
```

```
            ...
        } else { // if unlock amount requested (consumer)
            uint256 remaining = amount - unlocked; // determine remaining amount
            if (remaining > epoch.locked - epoch.unlocked) { // if remaining amount higher
then current lock amount
                unlocked += epoch.locked - epoch.unlocked; // unlock entire
                epoch.unlocked = epoch.locked;
                // missing update epochsState.unlocked
            } else {
                epoch.unlocked += remaining; // unlock only remaining
                break;
            }
        }
    }
    ...
}
```

UPDATES

- *Nov 03, 2023*: This issue has been acknowledged by the FCO team.

2.2.10. Should have events for `setServiceWallet()` and `setSignerWallet()` functions INFORMATIVE

Events are important in Solidity because they provide an efficient way to notify external applications of changes that occur on the blockchain. They allow contracts to emit messages to the blockchain network that can be detected and processed by other contracts or external services...

UPDATES

- *Nov 03, 2023*: This issue has been acknowledged and fixed by the FCO team.

2.2.11. Excessive variables and logic INFORMATIVE

Position:

- `PublicationHub.sol`#L65
- `PublicationHub.sol`#L83

Description:

The variable `internaCall` is declared but not used. The statement `eventEmitter.register()` has already been called inside the `__EventEmitter_init()` function, so it is not necessary to call it again. Therefore, both should be removed.

UPDATES

- *Nov 03, 2023*: This issue has been acknowledged and fixed by the FCO team.

2.2.12. Best practices to defend against reentrancy attacks INFORMATIVE

Position:

- `PublicationHub.sol`

Description:

- Consider using OpenZeppelin's ReentrancyGuard library, especially for functions with mutable state changes, to protect against reentrancy attacks.
- Or updating contract state completely before making any external calls.

UPDATES

- *Nov 03, 2023*: This issue has been acknowledged by the FCO team.

2.2.13. Avoid relying on `tx.origin` for authentication INFORMATIVE

Recommend fix

Positions:

- `FCOToken.sol`#L200
- `FCOToken.sol`#L276

Description:

Use `msg.sender` instead of `tx.origin` to check the caller's identity, as `tx.origin` can be manipulated by malicious contracts.

```
function use(address account, uint256 amount) public onlyOperator {
    _nonZeroAmount(amount, true);
    require(tx.origin == account, "Not allowed");
    ...
}

function approveWithSign(ApproveWithSignDat a calldata approveWithSignData) public {
    address account = approveWithSignData.data.account;
    require(tx.origin == account, "Bad tx origin for approve ws");
    ...
}
```

UPDATES

- *Nov 03, 2023*: This issue has been acknowledged by the FCO team.

3. VERSION HISTORY

Version	Date	Status/Change	Created by
1.0	*Nov 10, 2023*	Public Report	Verichains Lab
1.1	*Dec 20, 2023*	Public Report	Verichains Lab

Table 5. Report versions history